Chapter

15 *The CPP2SDL Tool*

The CPP2SDL tool is a C/C++-to-SDL translator that makes it possible to access C or C++ declarations in SDL. The tool takes a set of C/C++ header files as input and generates SDL declarations for a configurable set of the C/C++ declarations in these files.

CPP2SDL is the new generation of the H2SDL utility. Compared to its predecessor, CPP2SDL offers a comprehensive C++ support as well as superior translation configurability. CPP2SDL is fully integrated in Telelogic Tau SDL suite, but can also be executed as a stand-alone utility from the command shell.

This chapter is the reference manual for CPP2SDL. The reader is assumed to be familiar with C/C++ and SDL.

Introduction

The overall purpose of the CPP2SDL tool is to provide a convenient means of making external C or C++ declarations available in an SDL context. This is accomplished by translating the C/C++ declarations into representing SDL declarations. These resulting declarations can be injected at an arbitrary level in the SDL scope hierarchy, and may then be used just as if they actually were declared at that scope level. When target code is generated for the SDL system, the Code Generator produces C or C++ code for usages of generated SDL declarations that matches the original C/C++ declarations. The picture below depicts the data flow when using CPP2SDL, and the context of the tool.

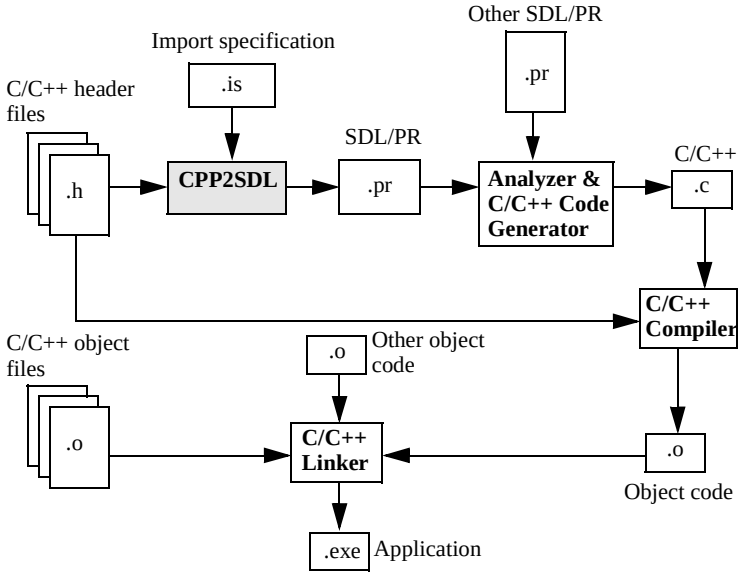


Figure 170 CPP2SDL Data Flow and Context

As can be seen in the figure, the input to CPP2SDL is a set of C/C++ header files and, optionally, an import specification. From this input CPP2SDL generates an SDL/PR file containing SDL representations for the declarations in the header files, or for a subset of these declarations according to what is specified in the import specification. The generated SDL/PR is analyzed together with other SDL/PR, e.g. the

SDL/PR for the SDL system. The Code Generator then generates target C/C++ code which is compiled by a C/C++ compiler. Note that the original C/C++ headers are used in this compilation. The resulting object code is linked together with the object files belonging to the C/C++ headers. Other object files are also included, e.g. the precompiled SDL kernel that is to be used. The result is an executable application.

CPP2SDL translates from C/C++ to SDL according to certain translation rules. These translation rules have been designed to be as simple and intuitive as possible. A user that is familiar with C/C++ should find it straight-forward to use a C/C++ declaration from SDL. The translation rules are described in full detail in "C/C++ to SDL Translation Rules" on page 778. Although CPP2SDL supports translation of a major part of the C and C++ languages, not everything is supported. The limitations of CPP2SDL are listed in "Known Limitations" on page 28 in chapter 2, *Release Notes*.

Executing CPP2SDL

Normally, CPP2SDL is automatically invoked by the SDL Analyzer as part of the make process. Input header files and tool options are then specified in the Organizer. However, CPP2SDL may also be executed as a stand-alone tool from a command shell, and in that case input headers and tool options are given as command-line options.

This section begins with a description of the integration with the Organizer and the Analyzer. Then how to execute CPP2SDL from the command-line is described. Finally, follows a section on how to run the tool through the PostMaster.

Execution from the Organizer

The most common way to execute CPP2SDL should be from the Organizer. In fact CPP2SDL will be started automatically by the Analyzer once for each import specification symbol it finds in the Organizer view (see "Import Specifications" on page 771 to learn about import specifications). The Analyzer executes CPP2SDL by means of the PostMaster as described in "Execution from the PostMaster" on page 769. All messages that are output during the execution will be printed in the Organizer Log Window.

Example 77: Executing CPP2SDL from the Organizer ----------

Consider a simple SDL system with one block and one process that needs to access some C++ declarations. At system level certain declarations of the C++ header file `general.h` is used, and at process level declarations of the files `f1.h` and `f2.h` are needed. Figure 171 below shows how the Organizer view of this SDL system could look like.

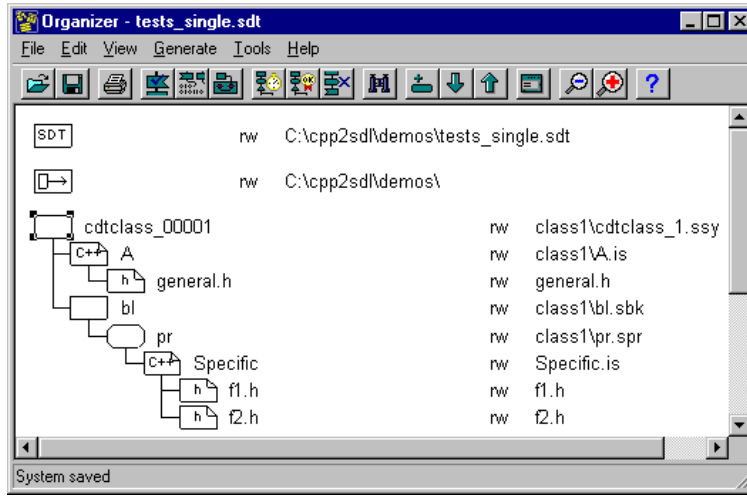


Figure 171 Organizer view with headers to be translated by CPP2SDL

When this system is analyzed, the Analyzer will execute CPP2SDL once for the file `general.h`, and once for the files `f1.h` and `f2.h`. The result of the first translation is a set of SDL declarations that are injected at system level, and thus will be accessible in all scopes. The result of the second translation is a set of SDL declarations that are injected at process level and thus are not accessible in the system or in the block scope.

Adding Import Specifications to the Organizer view

The first step in accessing C/C++ declarations from SDL is to insert a PR symbol at the place in the SDL specification where the C/C++ declarations are to be used. The PR symbol represents the inclusion of the SDL PR that is the translation of the C/C++ declarations.

To specify that this should be an import specification, double-click the PR symbol either in the Organizer or in the SDL Editor to open the Edit Document dialog. In the dialog it is possible to select either C Import Specification or C++ Import Specification.

An import specification can be edited manually by means of the Text Editor (see “Import Specifications” on page 771 to learn about import

specifications). However, an import specification can also be edited in the CPP2SDL Options dialog described below.

After adding an import specification it is necessary to specify which C/C++ header files are to be translated. This is done by selecting the import specification in the Organizer and then use the Add Existing and Add New commands to select or create C/C++ header files respectively.

Setting CPP2SDL Options in the Organizer

Required options to CPP2SDL may be specified in the Organizer for each import specification by using the CPP2SDL Options dialog. This dialog may be opened from the menu that appears when the right mouse button is pressed on an import specification symbol. Figure 172 shows this dialog.

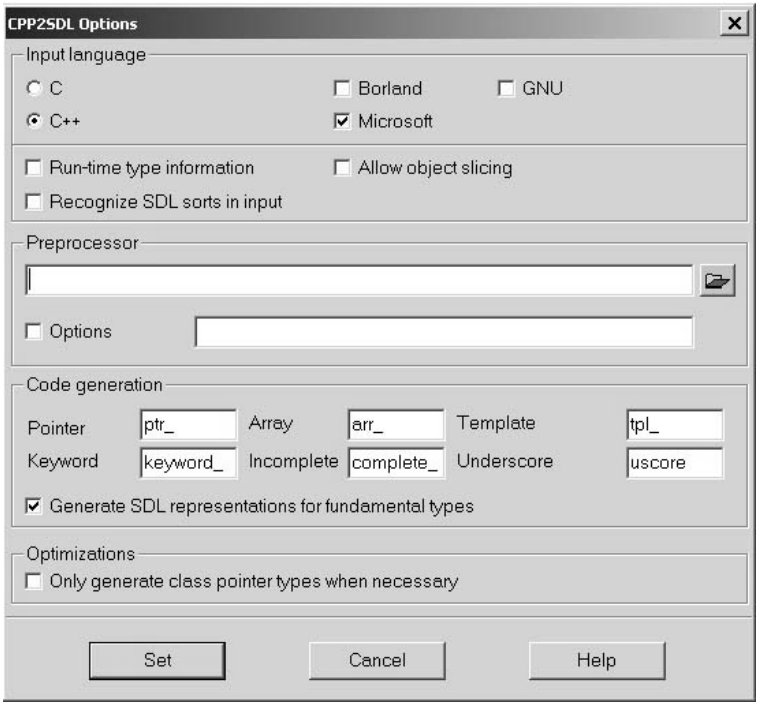


Figure 172 The CPP2SDL Options Dialog

The fields and buttons of the CPP2SDL Options dialog correspond directly to the command-line options described in "Command-Line Options" on page 764:

- *Language*

  These radio buttons select the input language. If C is selected, CPP2SDL will be executed in C mode, i.e. as with the `-c` command-line option.

- *Dialect*

  These check boxes determine what dialects to support in the input, and correspond to the `-dialects` command-line option. If no check-boxes are marked, the ANSI C/C++ dialect is supported.

- *Run-Time Type Information*

  This check box should be set if Run-Time Type Information (RTTI) is available in C++ and should be supported in the SDL translation. It corresponds to the `-rtti` command-line option.

- *Allow object slicing*

  This check box should be set if object slicing should be supported in the SDL translation. It corresponds to the `-slicing` command-line option.

- *Recognize SDL sorts in input*

  This check box should be set if SDL sorts should be recognized in the input. It corresponds to the `-sdlsorts` command-line option.

- *Preprocessor*

  This field is used to specify the preprocessor to use for preprocessing the input. It corresponds to the `-preprocessor` command-line option. This field also has a browse button that makes it possible to select the preprocessor from a file selection dialog.

- *Preprocessor options*

  This field should contain the options to the preprocessor. It corresponds to the `-cppoptions` command-line option.

- *Pointer, Array, Template, Keyword, Incomplete, Underscore*

These fields specify the prefixes and suffixes that are used when C/C++ names must be modified in the SDL translation. They correspond to the `-prefix` and `-suffix` command-line options.

- *Generate SDL representations for fundamental types*

  This check box should be set if SDL representations for fundamental C/C++ types should be included in the translation. It corresponds to the `-generatecpptypes` command-line option.

- *Only generate class pointer types when necessary*

  If this check box is set, CPP2SDL will optimize the generation of class pointer types. It corresponds to the `-optclasspointers` command-line option.

Execution from the Command-Line

CPP2SDL is invoked from the command-line by the command:

```
cpp2sdl [options] <C/C++ header files>
```

Unless the `-post` option is set, all messages that are output by the tool, e.g. errors and warnings, will be printed on the standard error stream (`stderr`).

CPP2SDL will translate the declarations in the specified C/C++ header files, or a subset of these declarations if a suitable import specification is used (see "Import Specifications" on page 771). The resulting SDL declarations will be saved in a file called *`name`*`.pr`, where *`name`* is the name of the import specification used. If no import specification is used, *`name`* will be the name of the first input header file.The output file will be placed in the same directory from where CPP2SDL is executed.

Command-Line Options

The command-line options recognized by CPP2SDL are listed and explained below. Note that an option may be abbreviated as indicated by the underlined part of the option name.

- `-`**`a`**`ppend`

  Append the generated SDL declarations to the file that is specified with the `-output` option. If that file does not exist, this option will

be ignored and CPP2SDL will create a new file for the output as usual.

- `-c`

  Execute in C mode. CPP2SDL will assume that no C++ specific constructs are encountered in the input headers. If this assumption does not hold, the result of the translation is undefined. See "Special Translation Rules for C Compilers" on page 839 for a detailed description of translation rule modifications that are caused by using this option.

- `-cppoptions <optionsstring>`

  Send the specified option string to the preprocessor. If the string contains white spaces, it must be quoted.

- `-dialects <dialect> <dialect> ... <dialect>`

  Accept the specified C/C++ dialects in the input headers. Supported dialects are

  – ANSI (ANSI C/C++)

  – BC (Borland C/C++)

  – GCC (Gnu C/C++)

  – MSVC (Microsoft Visual C/C++)

  – ALL (all supported C/C++ dialects)

  If this option is not used, CPP2SDL will assume that the input headers conform to the ANSI C/C++ dialect.

- `-errorlimit <number>`

  Set the maximum number of errors to report before terminating the translation. The default it to terminate when 5 errors have been found.

- -extsyn

  Will not generate for constants with numeric expressions, external synonyms with its value (if the expression can be calculated during translation). Default, i.e. without this option, the value is translated .

- `-generatecpptypes`

Include SDL representations for fundamental C/C++ types in the translation. See "SDL Library for Fundamental C/C++ Types" on page 841 for more information about what actually is generated when this option is used.

- `-help`

  Print a help message about CPP2SDL. No translation will be performed.

- `-importspecification <file>`

  Use the specified file as import specification for the translation. Import specifications are described in "Import Specifications" on page 771.

- `-nocheckinput`

  Do not check that all input headers are existing and readable before trying to translate them. The use of this option could make it easier to use CPP2SDL from scripts.

- `-nodepend`

  Do not translate depending declarations when using an import specification. Only the identifiers that are explicitly present in the import specification will be translated. If this option is set, CPP2SDL cannot guarantee that the resulting set of SDL declarations is complete and consistent. See "Import Specifications" on page 771 for more information.

- `-novariables`

  Do not generate external variables. This option is needed since the rules for where SDL allows declarations of external variables are more restrictive than for other declarations. For example, SDL does not allow external variables declared at system or block level. If this option is used, CPP2SDL will output a warning if it finds a construct that otherwise would be translated to an external variable.

- `-optclasspointers`

  Optimize the generation of class pointer types so that they are only generated when they appear in the input headers. If this option is not used, CPP2SDL will automatically generate a pointer type to all translated classes. Read more about this in "Classes, Structs and Unions" on page 796.

- `-output <file>`

  Write the resulting SDL declarations to the specified file. If the `-append` option is set, the result will be appended to the file. Otherwise a new file will be created, overwriting an existing file with the same name, if any.

  Note that all files that CPP2SDL generates will be placed in the same directory as the generated SDL/PR file.

- `-post`

  Start CPP2SDL as a PostMaster client waiting for requests from the PostMaster. The PostMaster messages that are handled by CPP2SDL are described in "Execution from the PostMaster" on page 769.

- `-prefix "ptr=<string> arr=<string> keyword=<string> incomplete=<string> tpl=<string>"`

  Use the specified name prefixes when generating SDL. CPP2SDL uses name prefixes when the original C/C++ names for some reason cannot be used in SDL. This option makes it possible to fully configure how such modified names are generated. This is often useful in order to avoid name clashes in SDL.

- `-preprocessor <executable>`

  Use the specified executable for preprocessing the input headers. The executable should be a preprocessor or C/C++ compiler that is supported by CPP2SDL:

  – 'cl' (Microsoft Visual C/C++ Compiler), **in Windows**.

  – 'cpp32' (Borland C/C++ Preprocessor), **in Windows**.

  – 'cpp' (C/C++ Preprocessor), **on Unix**.

  – 'cc' and 'CC' (Sun Workshop C and C++ Compilers), **on Unix**.

  – 'gcc' and 'g++' (GNU C and C++ Compilers), **on Unix**.

  If this option is not used, CPP2SDL will attempt to use 'cl' **in Windows**, and 'cpp' **on Unix**.

  Note that CPP2SDL uses name matching of the specified filename, with the file name extension stripped, to determine what preprocessor or compiler to use for preprocessing. If the specified name does

not match the name of any supported preprocessor or compiler on the current platform, CPP2SDL will attempt to call the executable like this:

```
<executable> <options> <input file> <output file>
```

`<options>` are the option string specified with the `-cppoptions` options.
If this call fails, CPP2SDL does not know how to preprocess the input headers and terminates.

Hint:

If you want to preprocess the input headers using a preprocessor that is not supported by CPP2SDL, you can write a simple shell script that wraps the call to the desired preprocessor. The script should conform to the call style that CPP2SDL uses for unknown preprocessors. Then execute CPP2SDL, using the `-preprocessor` option to specify the script as the preprocessor to use.

- `-ref`

  Include source references in the generated SDL. The format of these source references is described in "Source and Error References" on page 776.

- `-rtti`

  Assume Run-Time Type Information, and support dynamic casting. See "Run-Time Type Information and Dynamic Cast" on page 820 for more information what this means.

- `-sdlsorts`

  Recognize SDL sorts in input. CPP2SDL will translate C/C++ types that are prefixed with 'SDL_' to the corresponding SDL sort. Refer to "SDL Sorts in C/C++" on page 837 for an example on how this feature can be used.

- `-slicing`

  Generate SDL cast operators to support slicing of C++ objects. See "Type Compatibility between Inherited Classes" on page 816 for more information.

- `-sortmembers`

Sort struct members in SDL newtypes alphabetically.

- `-suffix "uscore=<string>"`

  Use the specified name suffixes in the generated SDL. CPP2SDL uses name suffixes when the original C/C++ name for some reason cannot be used in SDL. This option makes it possible to fully configure how such modified names are generated.

- `-targetdir <directory>`

  Set the target directory for generated files. CPP2SDL produces one single header file which includes all the header files that are to be translated. If this option is used, this generated header file is placed in the specified target directory. Otherwise the file will be placed in the same directory as the generated SDL/PR file.

- `-version`

  Show version information.

Example 78: Executing CPP2SDL from the command-line

```
% cpp2sdl -preprocessor /usr/ccs/lib/cpp -output
result.pr -prefix "ptr=p arr=a" -rtti -ref input.h
```

This command will translate the input header `input.h` to SDL and write the resulting SDL declarations to the file `result.pr`. The specified preprocessor 'cpp' will be used to preprocess the input. If the input contains pointer or array types, the corresponding SDL names will be prefixed with 'p' and 'a' respectively. Source references to the declarations in `input.h` will be generated by CPP2SDL, and Run-Time Type Information is assumed so that dynamic cast operators are generated.

Execution from the PostMaster

As mentioned above, CPP2SDL may be started as a PostMaster client by using the `-post` option at the command-line. As a PostMaster client, CPP2SDL will handle two different PostMaster events.

- SESTOP
- SECPP2SDLCOMMAND <optionstring>

The reception of a SESTOP event has the expected behavior; CPP2SDL ceases to be a PostMaster client and terminates.

The SECPP2SDLCOMMAND event has an option string as argument. The event will cause CPP2SDL to execute according to the options specified in that string. The format of the option string is the same as when CPP2SDL is executed from the command-line (see "Execution from the Command-Line" on page 764). All messages that are output by the tool will be broadcast to the PostMaster.

After the execution of a SECPP2SDLCOMMAND event a reply is sent:

SECPP2SDLCOMMANDREPLY <#errors> <#warnings> <status>

The <#errors> and <#warnings> arguments tell the number of errors and warnings that occurred during the translation, and <status> is a text string with the same information in a more readable form.

Example 79: Executing CPP2SDL from the PostMaster ––––––––––––

A single PostMaster may be started with this command:

```
% sdt -noclients
```

Then CPP2SDL is started as a PostMaster client:

```
% cpp2sdl -post &
```

CPP2SDL is now waiting for requests to come from the PostMaster. By using for example the SERVERPC application, events can be sent to it.

```
% serverpc 58000 58101 "-rtti -ref input.h"
```

58000 is the tool id of CPP2SDL, and 58101 is the event id for the SECPP2SDLCOMMAND event. As a result the following reply event could for example be received:

SECPP2SDLCOMMANDREPLY 0 2 "0 errors and 2 warnings"

Finally, CPP2SDL is terminated using the SESTOP event (id 58303):

```
% serverpc 58000 58303
```

––

Import Specifications

A simple but powerful way of configuring the translation of a set of C/C++ declarations is to use an import specification. As the name suggests, an import specification specifies how to import external code into SDL. An import specification for CPP2SDL is a textfile written in a simple C/C++-style syntax. The file consists of two sections that both are optional:

- `CPP2SDLOPTIONS`

  This section may contain options to the CPP2SDL tool. The syntax is the same as when CPP2SDL is executed as a stand-alone tool from the command-line. See "Command-Line Options" on page 764.

- `TRANSLATE`

  This section may contain a list of C/C++ identifiers. CPP2SDL will attempt to make these identifiers available in SDL by translating the corresponding declarations.

Options and identifiers in an import specification are delimited by newlines.

The example below shows a simple import specification where the identifiers `func`, `C` and `myint` are made available in SDL.

Example 80: A simple import specification ----------

```
CPP2SDLOPTIONS {
   -preprocessor /usr/ccs/lib/cpp
}

TRANSLATE {
   func
   C
   myint
   // C++ style comment (if supported by preproc.)
   /* C style comment */
}
```

The import specification file will be preprocessed by CPP2SDL with the same preprocessor and preprocessor options that are used when preprocessing the input C/C++ headers. This makes it possible to use, for instance, C/C++ comments and macros in an import specification.

Note:

Some preprocessors will refuse to preprocess files that have an unknown file name extension (for example `.is`). In that case the import specification file must be given a file name extension that is known to the preprocessor (for example `.h`).

If an identifier in an import specification refers to a declaration that depends on other declarations, CPP2SDL will, by default, translate all these depending declarations as well. This principle is applied recursively to all declarations that depend on depending declarations, thereby making sure that the resulting SDL declarations are complete and consistent. If the `-nodepend` option is set, depending declarations will not be translated automatically. Then the tool cannot guarantee that the resulting set of SDL declarations is complete and consistent.

Example 81: Translation of depending declarations ––––––––––––––

File `data.h`:

```
typedef int myint;

class C {
public:
  myint* mvar;
};
```

File `import.is`:

```
TRANSLATE {
   C
}
```

Execution of CPP2SDL from the command-line,

```
% cpp2sdl -importspecification import.is data.h
```

will produce resulting SDL declarations in the file `import.pr`:

```
SYNTYPE myint = int
ENDSYNTYPE myint;EXTERNAL 'C++';
NEWTYPE ptr_myint Ref( myint);
  OPERATORS
    ptr_myint : -> ptr_myint;
    ptr_myint : ptr_myint -> ptr_myint;/
ENDNEWTYPE ptr_myint;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
```

```
NEWTYPE C
  STRUCT
    mvar ptr_myint;
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

Here, the import specification only specifies that the class `C` shall be translated, but since the declaration of `C` depends on `myint*`, which in turn depends on `myint`, these declarations will be translated as well.

Note:

CPP2SDL will only translate those identifiers in an import specification that refer to declarations in namespaces (including the global namespace). If an identifier refers to another kind of declaration, for example a class member, it will be ignored and CPP2SDL will issue a warning.

Advanced Import Specifications

Besides from specifying which identifiers that should be translated to SDL, there are some more advanced constructs that may be used in an import specification.

Type Declarators

It is possible to append type declarators to identifiers that represent types. The same type declarators as in C/C++ are allowed, i.e. pointer (*), array ([]), and reference (&). Prefix and postfix declarators are separated by a dot (.).

Example 82: Type declarators in import specification ––––––––––

```
TRANSLATE {
   char*         // A pointer to char.
   MyClass.[8]   // An array of 8 MyClass.
   mytype&       // A reference to mytype.
   C*.[10]       // An array of 10 pointers to C.
}
```

Prototypes for Ellipsis Functions

The translation rule for a function with unspecified arguments (a.k.a an ellipsis function) requires that information is provided about which versions of the function that should be made available in SDL (see "Unspecified Arguments" on page 789). This information may be given in an import specification by specifying prototypes for the function.

Example 83: Prototypes for ellipsis functions in import specification

Input declaration:

```
int printf(const char*, ...);
```

Import specification:

```
TRANSLATE {
   printf
   printf(int)
   printf(double, char)
}
```

Resulting SDL declarations:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    printf : ptr_char, double, char -> int;
    printf : ptr_char, int -> int;
    printf : ptr_char -> int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';

NEWTYPE ptr_char Ref( char);
  OPERATORS
    ptr_char : -> ptr_char;
    ptr_char : ptr_char -> ptr_char;/
ENDNEWTYPE ptr_char;EXTERNAL 'C++';
```

Template Instantiations

Similar to ellipsis functions, the translation of templates requires additional information about how the templates should be instantiated (see "Templates" on page 831). This information may be specified in an import specification, using the same syntax as when templates are instantiated in C++.

Example 84: Template instantiations in import specification

Input declarations:

```
template <class C, int i> class S {
public:
  C arr[i];
};

template <class D> D func(const D& p1);
```

Import specification:

```
TRANSLATE {
   S<double, 5>
   func<unsigned int>
}
```

Resulting SDL declarations:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    tpl_func_unsigned_int /*#REFNAME 'func<unsigned
int >'*/ : unsigned_int -> unsigned_int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';

NEWTYPE arr_5_double CArray( 5, double);
ENDNEWTYPE arr_5_double;EXTERNAL 'C++';

NEWTYPE ptr_tpl_S_double_5 Ref( tpl_S_double_5);
  OPERATORS
    ptr_tpl_S_double_5 : -> ptr_tpl_S_double_5;
    ptr_tpl_S_double_5 : ptr_tpl_S_double_5 ->
    ptr_tpl_S_double_5;
ENDNEWTYPE ptr_tpl_S_double_5;EXTERNAL 'C++';

NEWTYPE tpl_S_double_5 /*#REFNAME 'S<double, 5 >'*/
  STRUCT
    arr arr_5_double;
  OPERATORS
    tpl_S_double_5 /*#REFNAME 'S'*/ : ->
    tpl_S_double_5;
   tpl_S_double_5 /*#REFNAME 'S'*/ : tpl_S_double_5
-> tpl_S_double_5;
ENDNEWTYPE tpl_S_double_5;EXTERNAL 'C++';
```

Note that since class template instantiations define types, it is possible to use type declarators for them.

Source and Error References

A source reference is a reference from a generated SDL declaration to the corresponding original C/C++ declaration. Source references are placed in the generated SDL/PR file.

An error reference is also a reference to a declaration in the input header file, but is used to point out an error (or a warning) in that file. Error references are therefore printed as messages to the standard error stream or to the Organizer Log Window.

CPP2SDL uses the #SDTREF format both for source and error references. See chapter 19, *SDT References* for more about #SDTREF.

Source References

When CPP2SDL is executed from the Organizer, or from the command-line with the `-ref` option set, the generated SDL/PR file will contain references to the input source files. Such a reference occurs just before a generated SDL declaration, and is on the form

```
/*#SDTREF(TEXT,filename,line)*/
```

where

- `filename` is the name of the input file where the corresponding C/C++ declaration can be found.
- `line` is the line number in that input file where the C/C++ declaration starts.

A source reference is shown in Example 85 below.

Example 85: Source references

```
/*#SDTREF(TEXT,input.h,226)*/
NEWTYPE S STRUCT
  a int;
OPERATORS
  get_a: S -> int;
ENDNEWTYPE S;EXTERNAL 'C++';
```

Error References

Error references have a similar format as source references but with a column position added after the line number:

```
/*#SDTREF(TEXT,filename,line,column)*/
```

CPP2SDL prints error references when errors or warnings are found during the translation. They are output to the standard error stream (`stderr`) or to the Organizer Log Window depending upon whether CPP2SDL is executed from the command-line or from the PostMaster.

Problems with error references may arise because of the preprocessor. Among other things, the preprocessor expands macros, and a typical problem is illustrated in Example 86 below.

Example 86: Error References

File `def.h`:

```
#define init InitializingFunction

void init(undefinedType *, int);
```

If CPP2SDL translates this file, the following error message will be printed:

```
#SDTREF(TEXT,def.h,3,27)
ERROR 3200 Syntax error.
```

Here the syntax error occurs at position (3,11) in the source file, but because of the macro expansion of `init` to `InitializingFunction`, CPP2SDL will report the error at position (3,27) instead. Thus the column position is several characters off the target in the original file. When using the Organizer Log's *Show Error* function (see "Show Error" on page 183 in chapter 2, *The Organizer*) to view the source of this error message, the cursor will be placed at `int` instead of at `undefinedType`. CPP2SDL calculates both source and error references from the preprocessed source code, and this may lead to reference problems when macros are involved.

C/C++ to SDL Translation Rules

The general idea behind the CPP2SDL tool is to take a set of C/C++ header files, preprocess them, and translate some or all of the declarations in these headers into SDL/PR representations. This section describes the rules for this translation process.

Each C/C++ construct is described in a subsection of its own. First, a general rule for the translation of the construct is presented. Then follows a description of exceptions to this rules, and rationals for these exceptions.

Before proceeding, it should be noted that the translation rules have been designed to support both C and C++ target compilers. To a large extent the translation rules are actually independent of whether a C or C++ target compiler is used. However, there are some differences, so when CPP2SDL executes in "C mode" (i.e. with the `-c` option set) a few translation rules are slightly modified. These modifications are described in "Special Translation Rules for C Compilers" on page 839.

Names

Rule: The name of a C/C++ identifier is the same in SDL.

The naming rules of identifiers in SDL and C/C++ are rather similar but differs in two important aspects:

- SDL is a case-insensitive language, while C/C++ is case-sensitive.
- SDL has some restrictions for how underscores may be used in names. C/C++ has no such restrictions.

To overcome these differences tool specific extensions have been made in the supported SDL dialect. The Analyzer has an option to handle case sensitive SDL (see "Set-Case-Sensitive" on page 2416 in chapter 55, *The SDL Analyzer*), and most of the restrictions with underscores have been removed. However, the rule that a name that ends with an underscore should be concatenated with the following name, makes it necessary to modify such names in the SDL mapping. This is done by appending a string suffix to such names. This string is by default "uscore" but may be configured to an arbitrary string be means of the CPP2SDL option
`-suffix`.

Another case where a name in C/C++ cannot be retained in the SDL translation is when the name is an SDL keyword. Such names are prefixed with a user-configurable string that by default is `keyword_`. The option `-prefix` can be used to configure this string.

Example 87 below gives some examples of the translation rules for names.

Example 87: Translation of names ––––––––––––––––––––––––––

C++:

```
int ABC, abc;   // Case sensitivity
char u__sc, _w, x_; // Unrestricted use of
underscores
double signal;  // SDL keyword
```

SDL:

```
DCL ABC int; EXTERNAL 'C++';
DCL abc int; EXTERNAL 'C++';
DCL u__sc char; EXTERNAL 'C++';
DCL _w char; EXTERNAL 'C++';
DCL x_uscore /*#REFNAME 'x_'*/ char; EXTERNAL 'C++';
DCL keyword_signal /*#REFNAME 'signal'*/ double;
EXTERNAL 'C++';
```

––

Note the #REFNAME directive that passes the original C/C++ name to the Code Generator for names that are modified in the SDL translation.

Fundamental Types

Rule: A fundamental C/C++ type is mapped to an SDL sort with the same name.

The SDL sorts that represent fundamental C/C++ types are not generated by CPP2SDL but are defined in special SDL/PR files that are included if the `-generatecpptypes` option is set. The SDL sorts in these files are normally syntypes of predefined SDL sorts. Refer to "SDL Library for Fundamental C/C++ Types" on page 841 for more information.

The table below shows how the fundamental C/C++ types are translated to SDL sorts, and what predefined SDL sort that correspond to these SDL sorts.

C/C++ Fundamental Type	SDL Sort	Predefined SDL Sort
signed int int	int	Integer
unsigned int unsigned	unsigned_int	Integer
signed long int signed long long int long	long_int	Integer
unsigned long int unsigned long	unsigned_long_int	Integer
signed short int signed short short int short	short_int	Integer
unsigned short int unsigned short	unsigned_short_int	Integer
signed long long int signed long long long long int long long	long_long_int	Integer
unsigned long long int unsigned long long	unsigned_long_long_int	Integer
char	char	Character
signed char	signed_char	Character
unsigned char	unsigned_char	Octet
wchar_t	wchar_t	N/A
float	float	Real
double long double	double	Real
bool	bool	Boolean
void	N/A	N/A

Note that the `wchar_t` type has no counterpart in SDL, and thus is represented by a newtype rather than a syntype of a predefined sort.

Also note that the special `void` type is not represented explicitly in SDL. Instead this type is translated by omitting input and result arguments to operators as described in "Functions" on page 786.

Type Declarators

There are two different type declarators in C; pointer (*), and array ([]). C++ has one additional type declarator; reference (&).

Pointers

Rule: A type with a pointer declarator is translated by applying the `Ref` generator on the SDL sort that corresponds to that type.

The name of the generated newtype for the pointer is prefixed with a user-configurable string that by default is `ptr_`. The option `-prefix` can be used to configure this string.

Untyped pointers (`void*`) are translated to a special SDL sort called `ptr_void`. See "SDL Library for Fundamental C/C++ Types" on page 841 for more information about the `Ref` generator and the special `ptr_void` sort.

Example 88: Translation of pointers

C++:

```
typedef int* p_int;
extern void* generalp;
```

SDL:

```
NEWTYPE ptr_int Ref( int);
  OPERATORS
    ptr_int : -> ptr_int;
    ptr_int : ptr_int -> ptr_int;
ENDNEWTYPE ptr_int;EXTERNAL 'C++';
SYNTYPE p_int = ptr_int
ENDSYNTYPE p_int;EXTERNAL 'C++';
DCL generalp ptr_void; EXTERNAL 'C++';
```

Arrays

Rule: A type with an array declarator is translated by applying the `CArray` generator on the SDL sort that corresponds to that type.

There is one important exception to this rule. Array declarators that do not specify the size of the array are translated in the same way as pointers (see "Pointers" on page 781).

The name of the generated newtype for an array type with a specified size is prefixed with a user-configurable string that by default is "arr_". The option `-prefix` can be used to configure this string. The name also contains the size of the array, since the size is used in the `CArray` generator instantiation and thus is significant in SDL. This makes SDL array sorts of different sizes type incompatible, but this is normally not a big problem since the elements of the arrays are type compatible.

Note:

SDL array sorts corresponding to C/C++ arrays with different sizes are normally type incompatible.

Example 89: Translation of arrays

C++:

```
extern char c_arr1[20];
extern char c_arr2[];
```

SDL:

```
NEWTYPE arr_20_char CArray( 20, char);
ENDNEWTYPE arr_20_char;EXTERNAL 'C++';
DCL c_arr1 arr_20_char; EXTERNAL 'C++';
NEWTYPE ptr_char Ref( char);
  OPERATORS
    ptr_char : -> ptr_char;
    ptr_char : ptr_char -> ptr_char;
ENDNEWTYPE ptr_char;EXTERNAL 'C++';
DCL c_arr2 ptr_char; EXTERNAL 'C++';
```

References

Rule: A type with a reference declarator is translated as normal, i.e. the reference declarator is not translated to SDL.

A C++ reference can be looked upon as a constant pointer that is automatically de-referenced each time it is used. This makes a reference an alternative name for an object. Since no difference will be made between an object and a reference to an object in the SDL mapping, references will appear to be objects in SDL.

Example 90: Translation of references

C++:

```
extern int i; /* i is initialized elsewhere */
extern int& r; /* r is initialized to i elsewhere
                                          (int& r =
i;), i.e. r and i refers to
                    the same int. */
```

SDL:

```
DCL i int; EXTERNAL 'C++';
DCL r int; EXTERNAL 'C++';/* N.B. C++ reference! */
```

Note that if `r` in this example is assigned a value in SDL, it is in fact the object that `r` refers to (i.e. `i`) that gets a new value. This could be confusing if only the SDL translation of `r` is considered, and to avoid this a comment is attached to the declaration of `r` that tells that it is a reference in C++.

References could also appear as specifiers for formal function arguments. Such arguments will be translated to operator arguments marked with the IN/OUT keyword if they are non-constant (see "Argument Passing and Return Value" on page 787).

Enumerated Types

Rule: An enumerated type is translated to a newtype with literals corresponding to the enum literals.

A special case is when the enumerated type has no literals. Such a type can be treated as an integer in C/C++, and is consequently translated to a syntype of `int`.

Example 91: Translation of enumerated types

C/C++:

```
enum {} v;
enum E2 {};
enum E1 {a, b, c=10};
```

SDL:

```
DCL v int; EXTERNAL 'C++';
SYNTYPE E2 = int
ENDSYNTYPE E2;EXTERNAL 'C++';
NEWTYPE E1
```

```
  LITERALS a, b, c;
  OPERATORS
    IntToEnum /*#REFNAME '(E1)'*/ : int -> E1;
    EnumToInt : E1 -> int; /*#OP(PY)*/
    ORDERING;
ENDNEWTYPE E1;EXTERNAL 'C++';
```

By using the "type conversion" operators `EnumToInt` and `IntToEnum` integer arithmetic and comparisons become available in SDL also for enumerations. As can be seen from the #REFNAME directive in the example above, the generated code for calls to the `IntToEnum` operator will be a C style cast from `int` to `enum`. This explicit type conversion should be acceptable by all target compilers. Also note that the #OP(PY) directive means that there will be no generated code for calls to the `EnumToInt` operator, which is desired since that type conversion is implicit in C/C++.

If the enumerated type is incomplete, i.e. if the enum tag is missing, the translation rule is slightly modified according to the translation rules for incomplete types (see "Incomplete Types" on page 823). For enumerations, these rules have the following impact:

· The name of the generated newtype follows the naming rules for incomplete types described in "Incomplete Types" on page 823.

· The newtype will not be external, since it does not correspond to a C/C++ type that may be referred to.

· The `IntToEnum` operator will not be generated for the same reason.

Typedef Declarations

Rule: A typedef declaration is translated to an SDL syntype declaration.

There are two exceptions to this rule:

- A typedef declaration of a tagged type[1], where the typedef name is the same as the name of the tag.
- A typedef declaration where the typedef name has been omitted. This is a legal but not very common case.

In these cases the typedef declarations do not define new typenames, and thus no syntypes need to be generated.

1. A tagged type is a class, struct, union or enum type with a tag.

Another special case is when a synonym for `void` is introduced by means of a typedef declaration. Such a typedef declaration is not translated, but the typedef name will be remembered. References to the typedef name will then be translated in the same way as `void` would have been translated in that context.

Example 92: Translation of typedef declarations ––––––––––––––––

C++:

```
typedef int MyInt;
typedef struct r {
                  int a;
} r; // Typedef name is the same as the tag name!
typedef struct s {
  MyInt a;
}; // Omitted typedef name - legal but rare!
typedef void myvoid;
typedef myvoid myvoid2;
myvoid f(myvoid2);
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    f :;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
SYNTYPE MyInt = int
ENDSYNTYPE MyInt;EXTERNAL 'C++';
NEWTYPE ptr_r Ref( r);
  OPERATORS
    ptr_r : -> ptr_r;
    ptr_r : ptr_r -> ptr_r;
ENDNEWTYPE ptr_r;EXTERNAL 'C++';
NEWTYPE r
  STRUCT
    a int;
  OPERATORS
    r : -> r;
    r : r -> r;
ENDNEWTYPE r;EXTERNAL 'C++';
NEWTYPE ptr_s Ref( s);
  OPERATORS
    ptr_s : -> ptr_s;
    ptr_s : ptr_s -> ptr_s;
ENDNEWTYPE ptr_s;EXTERNAL 'C++';
NEWTYPE s
  STRUCT
    a MyInt;
  OPERATORS
    s : -> s;
    s : s -> s;
```

```
ENDNEWTYPE s;EXTERNAL 'C++';
```

Note:

Typedefs of function types are not supported by CPP2SDL, and will not be translated to SDL.

Functions

Rule: A function prototype is translated to an SDL operator signature.

This rule is valid both for member and non-member functions. Operators that result from functions that are members of a class will be placed in the newtype that is the translation of that class. Operators that result from non-member functions will be placed in a special newtype called `global_namespace`[1].

Member functions are described in "Members" on page 800, and the rest of this section will focus on non-member functions.

Example 93: Translation of non-member functions

C++:

```
char myfunc1(char);
int myfunc1();
void myfunc2();
void myfunc2(int);
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    myfunc1 : char -> char;
    myfunc1 : -> int;
    myfunc2 :;
    myfunc2 : int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
```

Note that functions without input arguments or return value, will be translated to operators without input arguments or return value. Such operators are not allowed according to the SDL96 standard, but are accepted by the SDL Analyzer as a tool specific language extension.

1. This name indicates that the newtype represents the global scope in C/C++. In C++ terminology this scope is often called the global namespace.

Overloaded Functions

Rule: Overloaded functions are translated to overloaded SDL operators.

The semantics of overloaded functions in C++ differs slightly from the semantics of overloaded operators in SDL. For example, C++ allows overloading on constant arguments which is not possible in SDL. A C++ header file may therefore contain overloaded functions that cannot be translated to SDL. Normally this is not a problem since the C++ compiler resolves generated calls to these functions correctly anyway.

Example 94: Translation of overloaded functions

C++:

```
int f0();
int f0(double);
int f1(int&);
int f1(const int&);
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    f0 : -> int;
    f0 : double -> int;
    f1 : int -> int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
```

Argument Passing and Return Value

Rule: Function arguments that are passed by reference are translated to IN/OUT operator arguments in SDL.

There is one exception to this rule; arguments that are references to constants do not translate to IN/OUT arguments since C++ allows these arguments to take variables as well as constant values.

Example 95: Translation of function arguments and return value

C++:

```
int f1 (int p1, int &p2, const int &p3, const int
*p4, int *const p5);
int &f2();
const int &f3();
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    f1 : int, IN/OUT int, int, ptr_int, ptr_int ->
int;
    f2 : -> int;
    f3 : -> int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
NEWTYPE ptr_int Ref( int);
  OPERATORS
    ptr_int : -> ptr_int;
    ptr_int : ptr_int -> ptr_int;
ENDNEWTYPE ptr_int;EXTERNAL 'C++';
```

The example shows that information about constant arguments is lost in the SDL mapping. Also note that no difference will be made in SDL between functions that return data by value, data by reference, or constant data by reference.

Finally note that IN/OUT arguments to operators is a tool specific SDL extension.

Default Arguments

Rule: A function with default arguments are translated to several overloaded SDL operators.

This translation is reasonable given the fact that each C++ function with default arguments may be rewritten to an equivalent set of overloaded C++ functions.

Example 96: Translation of functions with default arguments

C++:

```
int func(int a, int b = 5, int c = 7);
int func(int a); // Ambiguous function!
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    func : int, int, int -> int;
    func : int, int -> int;
    func : int -> int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
```

In C++, ambiguities between overloaded functions are allowed provided that the functions are never called. SDL is, however, more strict, and

operator resolution is made from the declarations of the operators. The second version of `func` in the example above is therefore not accessible in the SDL mapping.

Unspecified Arguments

Rule: The translation of a function with unspecified arguments (a.k.a. an ellipsis function) requires the usage of an import specification that specifies the types of the unknown arguments.

See "Prototypes for Ellipsis Functions" on page 774 for information about how an import specification can be used to "expand" ellipsis functions.

Inline Functions

Rule: A function that is declared to be inline is translated as an ordinary function.

This is natural since the `inline` keyword on functions can be seen as a directive to the C++ compiler, which only affects the way that calls to these functions are generated. This is of course nothing that needs to be visible in SDL.

Example 97: Translation of inline functions ––––––––––––––––––––

C++:

```
inline int fac(int n){/*...*/};
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    fac : int -> int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
```

––

Function Pointers

Rule: A function pointer is translated to an untyped pointer in SDL, i.e. to `ptr_void`.

This translation rule makes it possible to represent a function pointer in SDL, but it is not possible to call the function that it points to, or to assign the address of another function to it. That has to be done with inline C/C++ code, for example by means of the #CODE operator.

Example 98: Translation of function pointers ––––––––––––––––––––

C++:

```
typedef int (*fp)(int, char);
fp g(double);
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    g : double -> fp;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
SYNTYPE fp = ptr_void
ENDSYNTYPE fp;EXTERNAL 'C++';
```

––

The special `ptr_void` sort is described in "SDL Library for Fundamental C/C++ Types" on page 841.

Scope Units

Rule: A C/C++ scope unit is translated to an SDL newtype.

Note that the global scope (known as the global namespace in C++) is also translated to an SDL newtype. This newtype is called `global_namespace` and is a container for all operators that are the translation of non-member or global functions in the program. Other global declarations are however placed directly in the SDL scope that is the context of the translation.

Example 99: Translation of the global namespace ––––––––––––––––

C++:

```
int i;
void op(unsigned int);
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    op : unsigned_int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
DCL i int; EXTERNAL 'C++';
```

––

The most important scope units that may be found in a C/C++ header file are:

- Namespaces

- Classes, structs and unions
- Template classes

In C++ these scope units may be nested to arbitrary depth, but since nested newtypes are not allowed in SDL, the translation of a nested scope unit will be a newtype that has a name that is prefixed with a scope qualification prefix. This prefix consists of the names of all enclosing scope units separated by underscores ("_").

Example 100: Translation of nested scope units ––––––––––––––––

C++:

```
class C {
public:
          int ci;
  class CC {
  public:
    int op();
  };
};
```

SDL:

```
NEWTYPE ptr_C_CC Ref( C_CC);
  OPERATORS
    ptr_C_CC : -> ptr_C_CC;
    ptr_C_CC : ptr_C_CC -> ptr_C_CC;
ENDNEWTYPE ptr_C_CC;EXTERNAL 'C++';
NEWTYPE C_CC /*#REFNAME 'C::CC'*/
  OPERATORS
    op : C_CC -> int;
    C_CC /*#REFNAME 'CC'*/ : -> C_CC;
    C_CC /*#REFNAME 'CC'*/ : C_CC -> C_CC;
ENDNEWTYPE C_CC;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    ci int;
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

––

Compare the name "C_CC" of the nested class CC in this example, with the fully qualified name "C::CC" of this class in C++. The latter name

is provided in a #REFNAME directive as information to the Code Generator.

Namespaces

Rule: A namespace is translated to a newtype that may not be instantiated in SDL.

Classes, structs, unions and template classes not only define scope units, but also types. They may thus be instantiated in for example variable declarations. Namespaces, on the other hand, are plain scope units and may not be instantiated. This is indicated in the SDL mapping by means of a Code Generator directive called #NOTYPE. This directive enables the Code Generator to catch attempts to instantiate newtypes that originates from namespaces.

Example 101: Translation of namespaces ----------

C++:

```
namespace N {
          const int ci;
  class CC {
  public:
    int op();
  };
  int f(char);
}
```

SDL:

```
SYNONYM N_ci /*#REFNAME 'N::ci'*/ int = EXTERNAL
'C++';
NEWTYPE ptr_N_CC Ref( N_CC);
  OPERATORS
    ptr_N_CC : -> ptr_N_CC;
    ptr_N_CC : ptr_N_CC -> ptr_N_CC;
ENDNEWTYPE ptr_N_CC;EXTERNAL 'C++';
NEWTYPE N_CC /*#REFNAME 'N::CC'*/
  OPERATORS
    op : N_CC -> int;
    N_CC /*#REFNAME 'CC'*/ : -> N_CC;
    N_CC /*#REFNAME 'CC'*/ : N_CC -> N_CC;
ENDNEWTYPE N_CC;EXTERNAL 'C++';
NEWTYPE N /*#NOTYPE*/
  OPERATORS
    N_f /*#REFNAME 'N::f'*/ : char -> int;
ENDNEWTYPE N;EXTERNAL 'C++';
```

Note that the newtype that corresponds to the namespace only contains the operators that are the translation of the functions declared in that namespace. This is analogous to how the global namespace is translated (see Example 99). Other declarations in the namespace will appear outside the newtype. All SDL declarations that are generated from namespace declarations will be prefixed with the name of the newtype that is the translation of that namespace. Also, their fully qualified C++ name is given in #REFNAME directives.

Variables

Rule: A variable is translated to an external variable if it is a non-member or global variable, or to a newtype field if it is a member variable.

Newtype fields that result from member variables of a class will be placed in the newtype that is the translation of that class. Member variables are described in "Members" on page 800, and the rest of this section will focus on non-member variables.

Example 102: Translation of non-member variables

C++:

```
int ivar, jvar;
class X {
        int j;
public:
  int Get() { return j;};
} xvar;
```

SDL:

```
DCL ivar int; EXTERNAL 'C++';
DCL jvar int; EXTERNAL 'C++';
NEWTYPE ptr_X Ref( X);
  OPERATORS
    ptr_X : -> ptr_X;
    ptr_X : ptr_X -> ptr_X;
ENDNEWTYPE ptr_X;EXTERNAL 'C++';
NEWTYPE X
  OPERATORS
    Get : X -> int;
    X : -> X;
    X : X -> X;
ENDNEWTYPE X;EXTERNAL 'C++';
DCL xvar X; EXTERNAL 'C++';
```

External variables are a tool-specific SDL extension that are similar to external synonyms.

External variables may only be declared in a process, procedure, service or operator diagram. Since CPP2SDL does not know the SDL context where the translation takes place, it has a command-line option called `-novariables` that tells whether external variables may be generated or not. When CPP2SDL is executed from the Organizer, this option is set automatically. If the option is set, and a C/C++ construct is found that would map to an external variable, CPP2SDL will print a warning.

Constants

Rule: A constant is translated to an external synonym.

This rule applies for all true C/C++ constants, i.e. constants that have been declared using the `const` type specifier. It is not uncommon, especially in older C API:s, to use macros to represent constants. Such constants will not be directly accessible in SDL since the preprocessor expands them before CPP2SDL begins the translation. However, simple macro constants may often be accessed by using inline target code, for example by means of the #CODE operator. As an alternative external synonyms could be declared to represent such macros.

Note: With the option -extsyn the translation of constants differs some. For constants with numeric expressions that can be calculated during translation, the default transformation rule is that also the constant value is translated. If -extsyn is switched on, translation is always an external synonym without its value.

Example 103Translation using -extsyn

C++:

```
const int FOO = 1;
const float ScoobieDoo = FOO/4;
const bool YOU;
```

SDL without -extsyn option (default behaviour):

```
SYNONYM FOO int = 1; EXTERNAL 'C++';
SYNONYM ScoobieDoo float = 0.25; EXTERNAL 'C++';
SYNONYM YOU bool = EXTERNAL 'C++';
```

SDL with -extsyn option:

```
SYNONYM FOO int = EXTERNAL 'C++';
```

```
SYNONYM ScoobieDoo float = EXTERNAL 'C++';
SYNONYM YOU bool = EXTERNAL 'C++';
```

Example 104: Translation of constants ––––––––––––––––––––––––

C++:

```
class MyClass;
const double pi = 3.1415;
const MyClass m(7, 'x');
```

SDL:

```
SYNONYM pi double = 3.1415; EXTERNAL 'C++';
NEWTYPE MyClass /*#NOTYPE*/
ENDNEWTYPE MyClass;EXTERNAL 'C++';
SYNONYM m MyClass = EXTERNAL 'C++';
```

––

Constant Expressions

Rule: Constant expressions are evaluated while translated to SDL.

Constant expressions may be encountered at a number of places in a C/C++ header, for example as constant initializers, or as size specifiers of array declarators or bitfields. If a constant expression has to be translated to SDL, CPP2SDL attempts to evaluate it during the translation in order to simplify its representation in SDL.

Example 105: Translation of constant expressions ––––––––––––––

C++:

```
enum e {a, b, c=10};
const int i = (2+c)*b;
struct s{
  int f1 : (2+c)*b;
};
typedef int intarr[sizeof(int)+1];
```

SDL:

```
NEWTYPE e
  LITERALS a, b, c;
  OPERATORS
    IntToEnum /*#REFNAME '(e)'*/ : int -> e;
    EnumToInt : e -> int; /*#OP(PY)*/
    ORDERING;
ENDNEWTYPE e;EXTERNAL 'C++';
SYNONYM i int = EXTERNAL 'C++';
```

```
NEWTYPE ptr_s Ref( s);
  OPERATORS
    ptr_s : -> ptr_s;
    ptr_s : ptr_s -> ptr_s;
ENDNEWTYPE ptr_s;EXTERNAL 'C++';
NEWTYPE s
  STRUCT
    f1 int : 12;
  OPERATORS
    s : -> s;
    s : s -> s;
ENDNEWTYPE s;EXTERNAL 'C++';
NEWTYPE arr_2_int CArray( 2, int);
ENDNEWTYPE arr_2_int;EXTERNAL 'C++';
SYNTYPE intarr = arr_2_int
ENDSYNTYPE intarr;EXTERNAL 'C++';
```

Note that not all the constant expressions in this example are visible in the SDL translation, and thus need not be evaluated by CPP2SDL.

Most constant expressions can be evaluated by CPP2SDL, but not all. In particular, expressions containing the `sizeof()` operator are difficult to evaluate since CPP2SDL has no information about what compiler that will be used to compile the generated C/C++ code. Some standard assumptions are therefore used when a `sizeof()` operator is encountered, and a warning will be issued to encourage manual inspection of the translation.

Classes, Structs and Unions

Rule: A class, struct or union is translated to an SDL newtype.

This rule follows from the fact that classes, structs and unions are scope units (see "Scope Units" on page 790).

This section mainly uses classes in the discussions and examples, but since the translation rules make no difference between classes, structs and unions, the same is valid for structs and unions. Example 106 shows the translation of an empty class, struct and union.

Example 106: Translation of classes, structs and unions

C++:

```
class C {};
struct S {};
union U {};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
NEWTYPE ptr_U Ref( U);
  OPERATORS
    ptr_U : -> ptr_U;
    ptr_U : ptr_U -> ptr_U;
ENDNEWTYPE ptr_U;EXTERNAL 'C++';
NEWTYPE U
  OPERATORS
    U : -> U;
    U : U -> U;
ENDNEWTYPE U;EXTERNAL 'C++';
```

In the example above three C++ types translate to six SDL sorts. The reason for this is that when CPP2SDL generates a newtype for a class, it will also, by default, generate a newtype that represents a pointer type for this class. This is convenient since pointers to a class often are needed. If the class inherits other classes this pointer newtype is in fact necessary, since it then holds cast operators to the base types of the class (see "Type Compatibility between Pointers to Inherited Classes" on page 817). If the command-line option `-optclasspointers` has been set, CPP2SDL will not generate this extra newtype unless a pointer to the class is explicitly present in the input code.

If the class, struct, or union has no tag, it is an incomplete type declaration. The translation rules for incomplete types are described in "Incomplete Types" on page 823.

Anonymous Unions

Rule: An anonymous union is translated by making its members become fields of the newtype that represents the enclosing scope unit.

This translation rule is natural since an anonymous union is no scope unit.

Example 107: Translation of Anonymous Unions ––––––––––––––––

C++:

```
struct S {
        int i;
  union {
    int j;
    int k;
  };
};
```

SDL:

```
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    j int;/* member of anonymous union */
    k int;/* member of anonymous union */
    i int;
  OPERATORS
    S : -> S;/* implicit parameter-less constructor
*/
    S : S -> S;/* implicit copy constructor */
ENDNEWTYPE S;EXTERNAL 'C++';
```

––

Note that an anonymous union is not an incomplete type declaration, although the syntax is similar. An anonymous union is not used to declare a type nor a variable, and does not define a type at all. Consequently, the translation rules for anonymous unions and incomplete types differ significantly. Compare with "Incomplete Types" on page 823.

Constructors

Rule: A constructor for a class is translated to an operator with the same name as the newtype that represents the class.

The return sort of the operator will be the sort defined by the newtype for the class, and the operator will of course also be placed in that newtype.

There are two different kinds of constructors in C++:

- User-defined constructors. These constructors are manually declared and implemented.
- Implicit constructors. These constructors are implicitly declared and are auto-generated by the C++ compiler, provided that they are not already declared by the user.

While a class may contain an arbitrary number of user-defined constructors, it may at the most contain two auto-generated ones; a parameter-less (or default) constructor and a copy constructor. A parameterless constructor is available only if the class has no user-defined constructors, and a copy constructor is available only if no user-defined copy constructor is declared.

CPP2SDL will generate operators both for user-defined and implicit constructors. Example 108 below shows a class with three user-defined constructors, and one implicit copy constructor.

Example 108: Translation of constructors ––––––––––––––––––––––

C++:

```
class C {
public:
  C();
  C(int i);
  C(char c);
  ~C();
};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    C : -> C;
    C : char -> C;
    C : int -> C;
    C : C -> C;/* implicit copy constructor */
```

```
ENDNEWTYPE C;EXTERNAL 'C++';
```

Destructors

Rule: A destructor is not translated to SDL.

The reason why a class destructor is not made accessible in SDL, is that it normally should not be called explicitly. Instead it will be called automatically when an object of the class goes out of scope or is deleted. See Example 108 for an example of how a destructor disappears in the SDL mapping.

Members

Rule: Member variables of a C++ class are translated to fields in the newtype that is the translation of that class, and member functions are translated to operators in the same newtype.

Other declarations than variables and functions in a class, for example type declarations, are also sometimes called members of the class, but they are not translated according to the translation rule above. Instead they are considered to be declarations on their own, but defined in an enclosing scope unit (i.e. the class). See "Scope Units" on page 790 for more information.

Example 109: Translation of class members

C++:

```
class C {
public:
  int mv1; // Member variable
  void mf1(long long p1); // Member function
  enum e {a,b,c}; // "Member" type declaration
};
```

SDL:

```
NEWTYPE C_e /*#REFNAME 'C::e'*/
  LITERALS a, b, c;
  OPERATORS
    IntToEnum /*#REFNAME '(C::e)'*/ : int -> C_e;
    EnumToInt : C_e -> int; /*#OP(PY)*/
    ORDERING;
ENDNEWTYPE C_e;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
```

```
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    mv1 int;
  OPERATORS
    mf1 : C, long_long_int;
    C : -> C;/* implicit parameter-less constructor
*/
    C : C -> C;/* implicit copy constructor */
ENDNEWTYPE C;EXTERNAL 'C++';
```

Note that the operator that represents a member function will have an additional initial formal argument. This argument has the sort of the newtype that represents the class where the member function is declared. Member functions are called from SDL in a functional style, where the first actual argument to the member function operator is the class instance on which the member function is to be invoked.

Member Access Specifier

Rule: Only members with public access specifier are translated to SDL.

This rule follows from the fact that public members of a class are the only members that are accessible from outside that class or its derived classes.

Example 110: Translation of members with different access specifiers

C++:

```
class C {
private:
        int i;
protected:
  int j;
public:
  int k;
  int GetI();
  int GetJ();
  int Calc (int x, int y);
};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
```

```
NEWTYPE C
  STRUCT
    k int;
  OPERATORS
    Calc : C, int, int -> int;
    GetI : C -> int;
    GetJ : C -> int;
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

Virtual Member Functions

Rule: A virtual member function is translated in the same way as an ordinary member function.

In C++, virtual functions of a base class may be redefined in derived classes. Although this means that there is only one version of a particular virtual function in a derived class, the version defined in the base class may still be called by means of explicit qualification. Both versions of the function must thus be present in the SDL translation, exactly as is the case for non-virtual functions. See "Inheritance" on page 807 for more about how C++ inheritance is represented in SDL.

Example 111: Translation of virtual member functions ––––––––––

C++:

```
class CPen {
public:
  virtual void Draw(); // Virtual member function
  double GetRep(); // Non-virtual member function
};
class CPenD : public CPen {
public:
  virtual void Draw(); // Redefinition of
CPen::Draw()
};
```

SDL:

```
NEWTYPE ptr_CPen Ref( CPen);
  OPERATORS
    ptr_CPen : -> ptr_CPen;
    ptr_CPen : ptr_CPen -> ptr_CPen;
ENDNEWTYPE ptr_CPen;EXTERNAL 'C++';
NEWTYPE CPen
  OPERATORS
    Draw : CPen;
    GetRep : CPen -> double;
```

```
    CPen : -> CPen;
    CPen : CPen -> CPen;
ENDNEWTYPE CPen;EXTERNAL 'C++';
NEWTYPE ptr_CPenD Ref( CPenD);
  OPERATORS
    cast : ptr_CPenD -> ptr_CPen; /*#OP(PY)*/
    ptr_CPenD : -> ptr_CPenD;
    ptr_CPenD : ptr_CPenD -> ptr_CPenD;
ENDNEWTYPE ptr_CPenD;EXTERNAL 'C++';
NEWTYPE CPenD
  OPERATORS
    Draw : CPenD;
    CPen_Draw /*#REFNAME 'CPen::Draw'*/ : CPenD;/*
Inherited from CPen */
    GetRep : CPenD -> double;/* Inherited from CPen
*/
    CPenD : -> CPenD;/
    CPenD : CPenD -> CPenD;
ENDNEWTYPE CPenD;EXTERNAL 'C++';
```

Pure Virtual Member Functions

Rule: A pure virtual member function is translated in the same way as an ordinary member function.

Although "pure virtuality" does not affect the translation of the member function itself, it will have impact on how the containing class, which is an abstract class, is translated. The reason is that special translation rules apply for abstract classes. See "Abstract Classes" on page 819 for more information and an example on how pure virtual member functions are translated.

Static Members

Rule: A static member is translated both as an ordinary member, and as if it was declared in the global namespace.

There will thus be two representations in SDL of a static C++ member. The additional representation is caused by the fact that a static member is accessible without having an instance of the class where it is defined.

As shown in Example 112 below, a static member variable will be translated both to a newtype field and an external variable (see "Variables" on page 793), while a static member function will result in both an operator in the newtype for the class and an operator in the special `global_namespace` newtype (see "Functions" on page 786).

Example 112: Translation of static members ––––––––––––––––––––

C++:

```
class C {
public:
  static int k;
  static void InitI(int);
};
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    C_InitI /*#REFNAME 'C::InitI'*/ : int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
DCL C_k /*#REFNAME 'C::k'*/ int; EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    k int;
  OPERATORS
    InitI : C, int;
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

––

If the resulting SDL declarations are to be inserted in an SDL context where external variables are not allowed (i.e. if CPP2SDL executes with the `-novariables` option set), static member variables cannot be translated to external variables. In that case only the standard translation of class members can be applied. Naturally, CPP2SDL will issue a warning if this happens.

Constant Members

Rule: A constant member is translated as an ordinary member, but with a #CONSTANT directive attached.

The semantics of a constant member variable is that it may not be written to after its initialization, and a constant member function may not change the state of its object. There is no way to express these restrictions in SDL, so the Analyzer will not be able to detect if they are violated. However, by attaching the #CONSTANT directive to SDL decla-

rations that result from constant members, the Code Generator can do the necessary checks.

Example 113: Translation of constant members

C++:

```
class C {
public:
  const int cm; // constant member
  C(int k) : cm(k) {};
  void Do(double);
                          void Undo(double) const; //
constant member function
};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    cm /*#CONSTANT*/ int;
  OPERATORS
    C : int -> C;
    Do : C, double;
    Undo : C, double; /*#CONSTANT*/
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

Member Constants

Rule: A member constant is translated both as an ordinary member with a #CONSTANT directive attached, and as if it was declared in the global namespace.

This translation rule is a combination of the translation rules for static and constant members, which is natural since a member constant is declared both to be constant and static in C++.

Example 114: Translation of member constants

C++:

```
class X {
public:
  static const int i = 99; // member constant
```

```
};
const int X::i; // definition of i
```

SDL:

```
SYNONYM X_i /*#REFNAME 'X::i'*/ int = EXTERNAL
'C++';
NEWTYPE ptr_X Ref( X);
  OPERATORS
    ptr_X : -> ptr_X;
    ptr_X : ptr_X -> ptr_X;
ENDNEWTYPE ptr_X;EXTERNAL 'C++';
NEWTYPE X
  STRUCT
    i /*#CONSTANT*/ int;
  OPERATORS
    X : -> X;
    X : X -> X;
ENDNEWTYPE X;EXTERNAL 'C++';
```

Mutable Member Variables

Rule: A mutable member variable is translated as an ordinary member variable.

The `mutable` keyword in C++ can be looked upon as some kind of compiler directive, and needs therefore not be visible in the SDL translation.

Bitfield Member Variables

Rule: A bitfield member variable is translated to an SDL bitfield.

This rule applies for all bitfields that have a name. Bitfields without name are not translated to SDL.

It would have been possible to translate bitfields to ordinary newtype fields. However, by including the bitfield size in the SDL translation, the Analyzer is given a possibility to check that these fields are not assigned values that will not fit in the corresponding bitfield.

Example 115: Translation of bitfields ----------

C++:

```
struct A {
  unsigned int i : 12;
  int : 3;
  bool dirty : 1;
};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  STRUCT
    dirty bool : 1;
    i unsigned_int : 12;
  OPERATORS
    A : -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
```

Note that bitfields are a tool-specific SDL extension.

Friends

Rule: Friend declarations will not be translated to SDL.

Friendship between a class C and another declaration D only affects what members of C that the implementation of D may access. It is therefore uninteresting to supply this information in the SDL translation of the class C.

Inheritance

Rule: C++ inheritance is represented in SDL by adding the translation of all public base class members to the newtype that represents a derived class.

This rule simply means that the C++ inheritance hierarchy is flattened in the SDL newtype representation. The reason for choosing this translation strategy, instead of using SDL inheritance between newtypes, is that the semantics of C++ and SDL inheritance is quite different.

All public member variables and member functions (but not constructors) of direct or indirect bases of a class will be generated in the newtype that is the translation of that class. Such an inherited field or operator will normally have the same name in SDL as in C++, but in some cases it is necessary to prefix the name with the name of the class from which it is inherited[1]. This happens when the name of the inherited member is the same as the name of one of the members in the derived

1. This naming rule is generalized in "Multiple Inheritance" on page 811.

class. Example 116 below shows how such ambiguities between inherited members are handled.

Example 116: Translation of inheritance ––––––––––––––––––––––

C++:

```
class A {
public:
  int am;
  A(char);
};
class B : public A {
public:
  char bm;
  virtual void calc();
  void set();
};
class C : public B {
public:
  int am;
  double cm;
  void calc(); // Redefines B::calc()
  void set();
};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  STRUCT
    am int;
  OPERATORS
    A : char -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
NEWTYPE ptr_B Ref( B);
  OPERATORS
    cast : ptr_B -> ptr_A; /*#OP(PY)*/
    ptr_B : -> ptr_B;
    ptr_B : ptr_B -> ptr_B;
ENDNEWTYPE ptr_B;EXTERNAL 'C++';
NEWTYPE B
  STRUCT
    am int;/* Inherited from A */
    bm char;
  OPERATORS
    calc : B;
    keyword_set /*#REFNAME 'set'*/ : B;
    B : B -> B;
```

```
ENDNEWTYPE B;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    cast : ptr_C -> ptr_A; /*#OP(PY)*/
    cast : ptr_C -> ptr_B; /*#OP(PY)*/
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    am int;
    B_am /*#REFNAME 'B::am'*/ int;/* Inherited from
A */
    bm char;/* Inherited from B */
    cm double;
  OPERATORS
    calc : C;
    B_calc /*#REFNAME 'B::calc'*/ : C;/* Inherited
from B */
    keyword_set /*#REFNAME 'set'*/ : C;
    B_keyword_set /*#REFNAME 'B::keyword_set'*/ :
C;/* Inherited from B */
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

In C++, it is always possible to use a fully qualified name when accessing a class member, even if the name of the member is unambiguous without qualification. In the example above, the member variable `bm` in `C` that is inherited from `B`, may be referred to both as `bm` and `B::bm`. To avoid getting too many fields and operators in the generated newtypes, only the unqualified name can be used from SDL. This is natural since qualification in C++ normally only is done when necessary to resolve ambiguities.

There are more cases where C++ allows a member to be accessed by more than one name, while the SDL translation only supplies one of these possible names. For example, this applies for inherited types and static members as shown in Example 117 below.

Example 117: Translation of inherited types and static members––––

C++:

```
class B {
public:
        static int mv;
  static char mf(double);
  struct s {
    int y;
  } ms;
```

```
};
class D : public B {
};
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    B_mf /*#REFNAME 'B::mf'*/ : double -> char;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
NEWTYPE ptr_B Ref( B);
  OPERATORS
    ptr_B : -> ptr_B;
    ptr_B : ptr_B -> ptr_B;
ENDNEWTYPE ptr_B;EXTERNAL 'C++';
NEWTYPE B
  STRUCT
    ms B_s;
    mv int;
  OPERATORS
    mf : B, double -> char;
    B : -> B;
    B : B -> B;
ENDNEWTYPE B;EXTERNAL 'C++';
DCL B_mv /*#REFNAME 'B::mv'*/ int; EXTERNAL 'C++';
NEWTYPE ptr_B_s Ref( B_s);
  OPERATORS
    ptr_B_s : -> ptr_B_s;
    ptr_B_s : ptr_B_s -> ptr_B_s;
ENDNEWTYPE ptr_B_s;EXTERNAL 'C++';
NEWTYPE B_s /*#REFNAME 'B::s'*/
  STRUCT
    y int;
  OPERATORS
    B_s /*#REFNAME 's'*/ : -> B_s;
    B_s /*#REFNAME 's'*/ : B_s -> B_s;
ENDNEWTYPE B_s;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    cast : ptr_D -> ptr_B; /*#OP(PY)*/
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  STRUCT
    ms B_s;/* Inherited from B */
    mv int;/* Inherited from B */
  OPERATORS
    mf : D, double -> char;/* Inherited from B */
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
```

The declarations `B::mv`, `B::mf` and `B::s` in this example may in C++ also be referred to by means of the names `D::mv`, `D::mf` and `D::s`. This is not possible in the SDL translation, i.e. there are no declarations called `D_mv`, `D_mf` or `D_s`. CPP2SDL will choose the first version of the names since the members are declared in `B`.

Multiple Inheritance

The translation rule for C++ inheritance works also when a class inherits from more than one base class. However, the naming strategy described in "Inheritance" on page 807 for handling ambiguous inherited members have to be generalized to also cover the case when a class inherits the same base class more than once.

Example 118: Translation of multiple inheritance

C++:

```
class A {
public:
  int m;
};
class B {
public:
  int m;
  int n;
};
class C: public A, public B {
};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  STRUCT
    m int;
  OPERATORS
    A : -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
NEWTYPE ptr_B Ref( B);
  OPERATORS
    ptr_B : -> ptr_B;
    ptr_B : ptr_B -> ptr_B;
ENDNEWTYPE ptr_B;EXTERNAL 'C++';
NEWTYPE B
  STRUCT
```

```
    m int;
    n int;
  OPERATORS
    B : -> B;
    B : B -> B;
ENDNEWTYPE B;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    cast : ptr_C -> ptr_B; /*#OP(PY)*/
    cast : ptr_C -> ptr_A; /*#OP(PY)*/
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  STRUCT
    A_m /*#REFNAME 'A::m'*/ int;/* Inherited from A
*/
    B_m /*#REFNAME 'B::m'*/ int;/* Inherited from B
*/
    n int;/* Inherited from B */
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

The names of the generated fields and operators correspond to the qualified names that must be used in C++ to access the members in question. The rule is to qualify an ambiguous member with the most specialized base class that makes the name of the member unique. In most cases this base class is the class where the ambiguous member is declared, but when the inheritance hierarchy forms a graph rather than a tree (see Example 119) it might be necessary to qualify with the name of a class further down on the inheritance path.

Note that in some extraordinary inheritance hierarchies, it is possible that a member of a base class is inaccessible in a derived class. This happens when the inherited member cannot be unambiguously qualified according to the naming rule described above. If this happens, CPP2SDL will not translate the member to SDL, and a warning will be printed.

Virtual and Non-Virtual Inheritance

Rule: Virtual inheritance is translated in the same way as ordinary inheritance.

Virtual inheritance affects the way data is replicated when multiple inheritance is used. As shown in "Multiple Inheritance" on page 811

members that are inherited more than once from the same base class need to be prefixed in SDL.

Since data from a base class is not replicated in derived classes that inherit from the base class with virtual inheritance, it would be possible to avoid prefixing the name of the members that are virtually inherited from the base class. However, since a virtually inherited member in general may be accessed using many alternative prefixes (corresponding to possible paths for reaching the member in the inheritance graph), and none of these prefixes can be said to be more natural to use than the others, all versions of the member's name are included in the SDL translation. This is the reason why no difference is made between virtual and non-virtual inheritance in SDL.

Example 119: Translation of virtual inheritance ––––––––––––––––––

C++:

```
class A {
public:
  int a;
};
class C : public A {
};
class D : public virtual A {
};
class E : public virtual A {
};
class G : public C, public D, public E {
};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  STRUCT
    a int;
  OPERATORS
    A : -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    cast : ptr_C -> ptr_A; /*#OP(PY)*/
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
```

```
NEWTYPE C
  STRUCT
    a int;/* Inherited from A */
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    cast : ptr_D -> ptr_A; /*#OP(PY)*/
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  STRUCT
    a int;/* Inherited from A */
  OPERATORS
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
NEWTYPE ptr_E Ref( E);
  OPERATORS
    cast : ptr_E -> ptr_A; /*#OP(PY)*/
    ptr_E : -> ptr_E;
    ptr_E : ptr_E -> ptr_E;
ENDNEWTYPE ptr_E;EXTERNAL 'C++';
NEWTYPE E
  STRUCT
    a int;/* Inherited from A */
  OPERATORS
    E : -> E;
    E : E -> E;
ENDNEWTYPE E;EXTERNAL 'C++';
NEWTYPE ptr_G Ref( G);
  OPERATORS
    cast : ptr_G -> ptr_E; /*#OP(PY)*/
    cast : ptr_G -> ptr_D; /*#OP(PY)*/
    cast : ptr_G -> ptr_C; /*#OP(PY)*/
    ptr_G : -> ptr_G;
    ptr_G : ptr_G -> ptr_G;
ENDNEWTYPE ptr_G;EXTERNAL 'C++';
NEWTYPE G
  STRUCT
    C_a /*#REFNAME 'C::a'*/ int;/* Inherited from A
*/
    D_a /*#REFNAME 'D::a'*/ int;/* Inherited from A
*/
    E_a /*#REFNAME 'E::a'*/ int;/* Inherited from A
*/
  OPERATORS
    G : -> G;
    G : G -> G;
ENDNEWTYPE G;EXTERNAL 'C++';
```

Inheritance Access Specifier

Rule: Only members that are inherited using public inheritance are translated to SDL.

When the inheritance is private or protected, the members of the base class are not accessible from outside the class. It is therefore natural to exclude members, that are inherited from private and protected bases, in the newtype that represents the derived class.

Example 120: Translation of inheritance with different access specifiers

C++:

```
class X {
public:
  int a;
  void f();
};
class Y1 : public X {};
class Y2 : protected X {};
class Y3 : private X {};
```

SDL:

```
NEWTYPE ptr_X Ref( X);
  OPERATORS
    ptr_X : -> ptr_X;
    ptr_X : ptr_X -> ptr_X;
ENDNEWTYPE ptr_X;EXTERNAL 'C++';
NEWTYPE X
  STRUCT
    a int;
  OPERATORS
    f : X;
    X : -> X;
    X : X -> X;
ENDNEWTYPE X;EXTERNAL 'C++';
NEWTYPE ptr_Y1 Ref( Y1);
  OPERATORS
    cast : ptr_Y1 -> ptr_X; /*#OP(PY)*/
    ptr_Y1 : -> ptr_Y1;
    ptr_Y1 : ptr_Y1 -> ptr_Y1;
ENDNEWTYPE ptr_Y1;EXTERNAL 'C++';
NEWTYPE Y1
  STRUCT
    a int;/* Inherited from X */
  OPERATORS
    f : Y1;/* Inherited from X */
    Y1 : -> Y1;
    Y1 : Y1 -> Y1;
ENDNEWTYPE Y1;EXTERNAL 'C++';
```

```
NEWTYPE ptr_Y2 Ref( Y2);
  OPERATORS
    ptr_Y2 : -> ptr_Y2;
    ptr_Y2 : ptr_Y2 -> ptr_Y2;
ENDNEWTYPE ptr_Y2;EXTERNAL 'C++';
NEWTYPE Y2
  OPERATORS
    Y2 : -> Y2;
    Y2 : Y2 -> Y2;
ENDNEWTYPE Y2;EXTERNAL 'C++';
NEWTYPE ptr_Y3 Ref( Y3);
  OPERATORS
    ptr_Y3 : -> ptr_Y3;
    ptr_Y3 : ptr_Y3 -> ptr_Y3;
ENDNEWTYPE ptr_Y3;EXTERNAL 'C++';
NEWTYPE Y3
  OPERATORS
    Y3 : -> Y3;
    Y3 : Y3 -> Y3;
ENDNEWTYPE Y3;EXTERNAL 'C++';
```

The inheritance access specifier also affects how casting from the derived type to the base type can be done. This is described in "Type Compatibility between Inherited Classes" on page 816 and in "Type Compatibility between Pointers to Inherited Classes" on page 817.

Type Compatibility between Inherited Classes

Rule: An object of a derived class may be assigned to an object of a base class by using an explicit cast operator in SDL.

The above assignment (known as slicing) is type-compatible in C++ without the use of a cast operator. Since only the common members are copied in the assignment, this operation is somewhat dangerous and is not generally recommended. Therefore, the cast operators that are needed in SDL to do slicing between objects, are only generated when the `-slicing` option is set.

Example 121: Generation of operators for slicing

C++:

```
class C {};
class D {};
class CD : public C, public D {};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
```

```
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  OPERATORS
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
NEWTYPE ptr_CD Ref( CD);
  OPERATORS
    cast : ptr_CD -> ptr_D; /*#OP(PY)*/
    cast : ptr_CD -> ptr_C; /*#OP(PY)*/
    ptr_CD : -> ptr_CD;
    ptr_CD : ptr_CD -> ptr_CD;
ENDNEWTYPE ptr_CD;EXTERNAL 'C++';
NEWTYPE CD
  OPERATORS
    cast : CD -> D; /*#OP(PY)*/
    cast : CD -> C; /*#OP(PY)*/
    CD : -> CD;
    CD : CD -> CD;
ENDNEWTYPE CD;EXTERNAL 'C++';
```

Note that the inheritance access specifier affects how these cast operators are generated. A cast operator from a class `D` to a class `B` will only be generated if `B` is a public unambiguous base of `D`. If it is private or protected, or is an ambiguous base for `D`, it is not allowed to cast from `D` to `B`.

Type Compatibility between Pointers to Inherited Classes

Rule: A pointer to an object of a derived class may be assigned to a pointer to an object of a base class by using an explicit cast operator in SDL.

The above assignment is type-compatible in C++, i.e. “up-casts” in a class hierarchy are implicit in C++. This is an important property of object-oriented languages that support for example polymorphism. In SDL, however, the newtypes for a base class and a derived class will be unrelated and thus type incompatible. To support up-casting in SDL,

explicit cast operators are generated in the newtype that represents the pointer type to a derived class.

Example 122: Generation of cast operators for up-casting ––––––––––

C++:

```
class C {};
class D {};
class CD : public C, public D {};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  OPERATORS
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
NEWTYPE ptr_CD Ref( CD);
  OPERATORS
    cast : ptr_CD -> ptr_D; /*#OP(PY)*/
    cast : ptr_CD -> ptr_C; /*#OP(PY)*/
    ptr_CD : -> ptr_CD;
    ptr_CD : ptr_CD -> ptr_CD;
ENDNEWTYPE ptr_CD;EXTERNAL 'C++';
NEWTYPE CD
  OPERATORS
    CD : -> CD;
    CD : CD -> CD;
ENDNEWTYPE CD;EXTERNAL 'C++';
```

––

Note that the inheritance access specifier is taken into consideration so that a cast operator from `Ref(D)` to `Ref(B)` only will be generated if the class `B` is a public unambiguous base of the class `D`.

Sometimes it is necessary to do down-casts, or even cross-casts, in a class hierarchy. Such casts (known as dynamic casts) are explicit both

in C++ and SDL. See "Run-Time Type Information and Dynamic Cast" on page 820 for more information.

Abstract Classes

Rule: An abstract class is translated to a newtype without constructor operators.

This translation rule makes it possible to declare pointers to an abstract class, but no objects of such a class may be allocated since there are no constructor operators that can be used as argument to the `new` operator (see "Dynamic Memory Management" on page 825).

Example 123: Translation of abstract classes

C++:

```
class C {
public:
  virtual int f(int) = 0; // pure virtual member
function
  C() {};
};
class D : public C {
};
```

SDL:

```
NEWTYPE ptr_C Ref( C);
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
  OPERATORS
    f : C, int -> int;
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    cast : ptr_D -> ptr_C; /*#OP(PY)*/
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  OPERATORS
    f : D, int -> int;/* Inherited from C */
ENDNEWTYPE D;EXTERNAL 'C++';
```

Note from the example above that abstractness is inherited to a derived class if not each pure virtual member function of all its base classes are redefined in the derived class.

Run-Time Type Information and Dynamic Cast

Rule: A pointer to an object of a base class may be assigned to a pointer to an object of a derived class, or to a pointer to an object of a base class of such a derived class, by using an explicit cast operator in SDL.

The above assignments require a dynamic cast in C++, which is done with an explicit cast operator that supports down-casts and cross-casts in an inheritance hierarchy. Since these casts require run-time type information (RTTI) about the dynamic type of an object, most C++ compilers have an option that must be set to safely support dynamic casts. For the same reason, CPP2SDL also has such an option called `-rtti`. If it is set, cast operators will be generated that enable the type conversions that are possible in C++ using dynamic casts.

The source type of a dynamic cast must be polymorphic, i.e. contain one or more virtual member functions, possibly inherited ones. For each such polymorphic class `X`, cast operators will be generated that convert from `Ref(X)` to `Ref(Y)`, for each class `Y` that either inherits from `X` (down-casts), or is a public base class of a class that inherits from `X` (cross-casts).

Example 124 below illustrates this translation rule. It is assumed that all classes in the example contain virtual functions and thus are polymorphic.

Example 124: Generation of cast operators for dynamic casting ––––

C++:

```
class A {};
class B: public A {};
class E {};
class D: protected E {};
class C: public B, public D {};
```

SDL:

```
NEWTYPE ptr_A Ref( A);
  OPERATORS
    ptr_A : -> ptr_A;
    ptr_A : ptr_A -> ptr_A;
ENDNEWTYPE ptr_A;EXTERNAL 'C++';
NEWTYPE A
  OPERATORS
    A : -> A;
    A : A -> A;
ENDNEWTYPE A;EXTERNAL 'C++';
NEWTYPE ptr_B Ref( B);
```

```
  OPERATORS
    cast /*#REFNAME 'dynamic_cast<B*>'*/ : ptr_A ->
ptr_B;
    cast : ptr_B -> ptr_A; /*#OP(PY)*/
    ptr_B : -> ptr_B;
    ptr_B : ptr_B -> ptr_B;
ENDNEWTYPE ptr_B;EXTERNAL 'C++';
NEWTYPE B
  OPERATORS
    B : -> B;
    B : B -> B;
ENDNEWTYPE B;EXTERNAL 'C++';
NEWTYPE ptr_E Ref( E);
  OPERATORS
    ptr_E : -> ptr_E;
    ptr_E : ptr_E -> ptr_E;
ENDNEWTYPE ptr_E;EXTERNAL 'C++';
NEWTYPE E
  OPERATORS
    E : -> E;
    E : E -> E;
ENDNEWTYPE E;EXTERNAL 'C++';
NEWTYPE ptr_D Ref( D);
  OPERATORS
    ptr_D : -> ptr_D;
    ptr_D : ptr_D -> ptr_D;
ENDNEWTYPE ptr_D;EXTERNAL 'C++';
NEWTYPE D
  OPERATORS
    D : -> D;
    D : D -> D;
ENDNEWTYPE D;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    cast /*#REFNAME 'dynamic_cast<A*>'*/ : ptr_D ->
ptr_A;
    cast /*#REFNAME 'dynamic_cast<B*>'*/ : ptr_D ->
ptr_B;
    cast /*#REFNAME 'dynamic_cast<D*>'*/ : ptr_A ->
ptr_D;
    cast /*#REFNAME 'dynamic_cast<D*>'*/ : ptr_B ->
ptr_D;
    cast /*#REFNAME 'dynamic_cast<C*>'*/ : ptr_D ->
ptr_C;
    cast : ptr_C -> ptr_D; /*#OP(PY)*/
    cast /*#REFNAME 'dynamic_cast<C*>'*/ : ptr_A ->
ptr_C;
    cast : ptr_C -> ptr_A; /*#OP(PY)*/
    cast /*#REFNAME 'dynamic_cast<C*>'*/ : ptr_B ->
ptr_C;
    cast : ptr_C -> ptr_B; /*#OP(PY)*/
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
NEWTYPE C
```

```
  OPERATORS
    C : -> C;
    C : C -> C;
ENDNEWTYPE C;EXTERNAL 'C++';
```

Note that no cast operators are generated that cast to `ptr_E` since `E` is a protected base of `D`. But in fact there are no cast operators that cast from `ptr_E` neither. The somewhat subtle reason for this is that those operators cannot be used in practice, since the protected inheritance makes it impossible to have a variable with `ptr_E` as static type and `ptr_C` as dynamic type. CPP2SDL will therefore not generate these cast operators.

In C++ dynamic casts work both for pointers and references to objects. In SDL, however, it is only possible to do dynamic casts between pointers, since references are not explicitly represented in the translation (see "References" on page 782).

Forward Declarations

Rule: A forward declaration is not translated to SDL.

This rule is valid for all forward declarations for which there are definitions later on in the header file. This is the most common case, and the purpose of such forward declarations is simply to make an identifier known to the C/C++ compiler so that it may be used before it is defined.

However, it is possible to make a forward declaration for which no definition exists in the header file. In that case CPP2SDL must generate an extra newtype to represent the missing definition. Since this extra newtype does not correspond to a real C/C++ type, it is marked with a #NOTYPE directive.

Example 125 below contains two forward declarations, one of which has no corresponding definition (`class C`).

Example 125: Translation of forward declarations

C++:

```
typedef struct S *fwdS;
class C *fwdC;
struct S {
        int a;
};
```

SDL:

```
SYNTYPE fwdS = ptr_S
ENDSYNTYPE fwdS;EXTERNAL 'C++';
NEWTYPE C /*#NOTYPE*/
ENDNEWTYPE C;EXTERNAL 'C++';
NEWTYPE ptr_C Ref( C);
  OPERATORS
    ptr_C : -> ptr_C;
    ptr_C : ptr_C -> ptr_C;
ENDNEWTYPE ptr_C;EXTERNAL 'C++';
DCL fwdC ptr_C; EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    a int;
  OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
```

Incomplete Types

Rule: An incomplete type declaration is translated to a newtype that may not be instantiated in SDL. This rule applies to all incomplete types even if they are declared within complete types. For example a tagless type within a container type will not be correctly instantiated in SDL.

Compare this translation rule with the one for namespaces (see "Namespaces" on page 792). While a namespace does not define a type at all, an incomplete type declaration defines an incomplete type that may not be referred to. That is the reason why such a newtype must not be instantiated in SDL.

C/C++ allows declarations of incomplete classes, structs, unions and enums, i.e. all types having a tag. Incomplete types are therefore also called tag-less types.

Incomplete types can be used in

- data declarations (i.e. variables, constants etc.)
- type declarations (i.e. typedefs)
- "useless" declarations (i.e. without declaring data or type)

Example 126: Translation of incomplete types ––––––––––––––––––

C++:

```
struct S {
  int i;
  struct {
    int j;
  } ss1, *ss2, ss3[2]; // Data declarations
};
typedef enum {
  a, b, c
} ss1, *ss2, ss3[2]; // Type declarations
typedef struct {
        int i;
}; // Missing type name - "useless" declaration
struct {
        int i;
}; // Missing variable name - "useless" declaration
```

SDL:

```
NEWTYPE S_incomplete_ss3
  STRUCT
    j int;
ENDNEWTYPE S_incomplete_ss3;
NEWTYPE ptr_S_incomplete_ss3 Ref( S_incomplete_ss3);
  OPERATORS
    ptr_S_incomplete_ss3 : -> ptr_S_incomplete_ss3;
    ptr_S_incomplete_ss3 : ptr_S_incomplete_ss3 ->
    ptr_S_incomplete_ss3;
ENDNEWTYPE ptr_S_incomplete_ss3;EXTERNAL 'C++';
NEWTYPE arr_2_S_incomplete_ss3 CArray(2,
S_incomplete_ss3);
ENDNEWTYPE arr_2_S_incomplete_ss3;EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    i int;
    ss1 S_incomplete_ss3;
    ss2 ptr_S_incomplete_ss3;
    ss3 arr_2_S_incomplete_ss3;
  OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
NEWTYPE incomplete_ss3
  LITERALS a, b, c;
  OPERATORS
    EnumToInt : incomplete_ss3 -> int; /*#OP(PY)*/
    ORDERING;
```

```
ENDNEWTYPE incomplete_ss3;
SYNTYPE ss1 = incomplete_ss3
ENDSYNTYPE ss1;EXTERNAL 'C++';
NEWTYPE ptr_incomplete_ss3 Ref( incomplete_ss3);
  OPERATORS
    ptr_incomplete_ss3 : -> ptr_incomplete_ss3;
    ptr_incomplete_ss3 : ptr_incomplete_ss3 ->
    ptr_incomplete_ss3;

ENDNEWTYPE ptr_incomplete_ss3;EXTERNAL 'C++';
SYNTYPE ss2 = ptr_incomplete_ss3
ENDSYNTYPE ss2;EXTERNAL 'C++';
NEWTYPE arr_2_incomplete_ss3 CArray( 2,
incomplete_ss3);
ENDNEWTYPE arr_2_incomplete_ss3;EXTERNAL 'C++';
SYNTYPE ss3 = arr_2_incomplete_ss3
ENDSYNTYPE ss3;EXTERNAL 'C++';
```

As can be seen in the example, incomplete types that are used in data or type declarations will have the name of the last declared variable or type, prefixed with a user-configurable string that by default is "incomplete_". The option `-prefix` can be used to configure this string.

Incomplete types in "useless" declarations will not be translated to SDL, and CPP2SDL will issue warnings that the declarations were ignored.

Note that the translation of incomplete enum declarations differs from the normal translation rule of an enum declaration. The differences are listed in "Enumerated Types" on page 783.

Finally note that incomplete classes, structs and unions define scope units although they are incomplete. Their names thus follows the rules for naming of scope units described in "Scope Units" on page 790.

Dynamic Memory Management

Rule: The C/C++ primitives for dynamic memory management is represented in SDL by means of special operators.

Dynamic memory management is done differently in C and C++. How C or C++ data is dynamically allocated and deallocated in SDL therefore depends on whether CPP2SDL executes in C or C++ mode (controlled by the option `-c`). In both cases dynamic memory management is done by means of special SDL operators that are defined in the `Ref` generator. However, the definition of the `Ref` generator is different in C

and C++ mode (see "SDL Library for Fundamental C/C++ Types" on page 841).

C Mode

The following operators are used to support dynamic memory management of C data from SDL:

- `Make!`
  Enables dynamic allocation of simple C data.
- `free`
  Enables dynamic deallocation of data that was allocated by the `Make!` operator.

Example 127: Dynamic memory management of C data from SDL ––––

C:

```
struct S {
  int i;
  double j;
};
```

SDL:

```
NEWTYPE ptr_S Ref( S);
ENDNEWTYPE ptr_S;EXTERNAL 'C';
NEWTYPE S /*#REFNAME 'struct S'*/
  STRUCT
    i int;
    j double;
ENDNEWTYPE S;EXTERNAL 'C';
```

SDL Usage:

```
dcl var s, ptrs ptr_s;
task {
  ptrs := (. var .);
  ptrs*>!i := 4;
  free(ptrs);
};
```

––

C++ Mode

The following operators are used to support dynamic memory management of C++ data from SDL:

- `new`[1]
  Enables dynamic allocation of scalar C++ data of, for example, class type, fundamental type, or pointer type. It corresponds to the C++ operator with the same name.

- `delete`
  Enables dynamic deallocation of data that was allocated by the `new` operator. It corresponds to the C++ operator with the same name.

- `new_array`
  Enables dynamic allocation of arrays of C++ data of, for example, class type, fundamental type, or pointer type. It corresponds to the C++ operator `new[]`.

- `delete_array`
  Enables dynamic deallocation of data that was allocated by the `new_array` operator. It corresponds to the C++ operator `delete[]`.

Example 128: Dynamic memory management of C++ data from SDL–

C++:

```
struct S {
  int i;
  double j;
};
```

Import Specification:

```
TRANSLATE {
  S**
  int*
}
```

SDL:

```
NEWTYPE ptr_int Ref( int);
  OPERATORS
    ptr_int : -> ptr_int;
    ptr_int : ptr_int -> ptr_int;
ENDNEWTYPE ptr_int;EXTERNAL 'C++';
NEWTYPE ptr_ptr_S Ref( ptr_S);
```

1. In fact the `Make!` operator can also be used in C++ mode. In that case it behaves exactly like the `new` operator.

```
  OPERATORS
    ptr_ptr_S : -> ptr_ptr_S;
    ptr_ptr_S : ptr_ptr_S -> ptr_ptr_S;
ENDNEWTYPE ptr_ptr_S;EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;'
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    i int;
    j double;
 OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
```

SDL Usage:

```
dcl ptrs ptr_s, ptrptrs ptr_ptr_s, ptri ptr_int;
task {
  ptrs := new(s);
  ptrs*>!i := 4;
  ptrptrs := new(ptr_s);
  ptri := new(int);
  delete(ptrs);
  delete(ptrptrs);
  delete(ptri);
  ptri := newArray(int, 5);
  deleteArray(ptri);
};
```

The input to the `new` and `new_array` operators must be an operator that corresponds to a constructor in C++. To enable dynamic allocation of data with non-class types, there must thus exist constructor operators for these types. These operators correspond to the implicit parameter-less and copy constructors, which exist for each C++ type. The definition of these constructor operators are part of the non-generated SDL declarations that are included when the `-generatecpptypes` option is set (see "SDL Library for Fundamental C/C++ Types" on page 841).

Overloaded Operators

Rule: An overloaded C++ operator is translated to a corresponding overloaded SDL operator.

Since the sets of operators that may be overloaded are different in C++ and SDL, not all overloaded C++ operators can be translated to SDL.

The table below shows which C++ operators that can be translated to SDL

:

C++ operator	Description	SDL operator
+	(binary) Addition	+
-	(binary) Subtraction	-
*	(binary) Multiplication	*
*	(unary prefix) Dereference	*>
/	(binary) Division	/
%	(binary) Modulo	rem
!	(unary prefix) Not	not
<	(binary) Less	<
>	(binary) Greater	>
<<	(binary) Left Shift	<
>>	(binary) Right Shift	>
==	(binary) Equal	=
!=	(binary) Not Equal	/=
<=	(binary) Less Equal	<=
>=	(binary) Greater Equal	>=
&&	(binary) And	and
||	(binary) Or	or

Note that even if there is a corresponding SDL operator to translate to, a C++ operator could be declared in a way that would make it necessary to qualify its SDL name, which is not possible. This happens for example if the operator is declared to be static, or declared inside a namespace (see "Scope Units" on page 790 for more about scope name qualifications). It may also happen due to name qualification rules for inherited members (see "Multiple Inheritance" on page 811). CPP2SDL will issue a warning if it encounters an overloaded operator that cannot be translated.

The table above shows that the translation of the less and greater operators (<, >) is the same as the translation of the shift operators (<<, >>). Obviously, this may lead to ambiguities if both these operator pairs are overloaded in a class. In that case, the less and greater operators will have precedence, and CPP2SDL will issue a warning that the overloaded shift operators cannot be translated to SDL.

Example 129: Translation of overloaded operators

C++:

```
class ostream {
public:
  ostream& operator<(const char* p1);
  ostream& operator<<(const char* p1);
  ostream& operator>>(const char* p1);
  static int operator%(int p1);
  bool operator!();
};
```

SDL:

```
NEWTYPE ptr_char Ref( char);
  OPERATORS
    ptr_char : -> ptr_char;
    ptr_char : ptr_char -> ptr_char;
ENDNEWTYPE ptr_char;EXTERNAL 'C++';
NEWTYPE ptr_ostream Ref( ostream);
  OPERATORS
    ptr_ostream : -> ptr_ostream;
    ptr_ostream : ptr_ostream -> ptr_ostream;
ENDNEWTYPE ptr_ostream;EXTERNAL 'C++';
NEWTYPE ostream
  OPERATORS
    "not" /*#REFNAME 'operator!'*/ : ostream ->
bool;
    "rem" /*#REFNAME 'operator%'*/ : ostream, int ->
int;
    "<" /*#REFNAME 'operator<'*/ : ostream, ptr_char
-> ostream;
    ostream : -> ostream;
    ostream : ostream -> ostream;
ENDNEWTYPE ostream;EXTERNAL 'C++';
```

Conversion Operators

Rule: A conversion operator is translated to a special `conv` operator in SDL.

In C++, a conversion operator is implicitly called by the compiler when a matching type conversion is needed. The `conv` operator, however, must be called explicitly in SDL.

Example 130: Translation of conversion operators ––––––––––––––

C++:

```
class Tiny {
public:
  operator int(); // Implicit conversion from Tiny
to int
};
```

SDL:

```
NEWTYPE ptr_Tiny Ref( Tiny);
  OPERATORS
    ptr_Tiny : -> ptr_Tiny;
    ptr_Tiny : ptr_Tiny -> ptr_Tiny;
ENDNEWTYPE ptr_Tiny;EXTERNAL 'C++';
NEWTYPE Tiny
  OPERATORS
    conv : Tiny -> int; /*#OP(PY)*/
    Tiny : -> Tiny;
    Tiny : Tiny -> Tiny;
ENDNEWTYPE Tiny;EXTERNAL 'C++';
```

––

Note that the `conv` operator returns the target type of the type conversion specified by the conversion operator. The #OP(PY) directive tells the Code Generator that the operator is implicitly called in C++.

Templates

Rule: A template declaration is translated to SDL by instantiating it.

A template declaration as such cannot be translated to SDL. Only specified instantiations of the template can be translated. CPP2SDL will print a warning about this when a template declaration is encountered.

A template instantiation may of course be present in the input headers. In that case CPP2SDL will translate the template instantiation by substituting all formal template arguments in the template declaration with the actual template arguments used in the template instantiation. If the input headers contain no suitable instantiation of a certain template, an import specification may be used to provide such an instantiation. See "Template Instantiations" on page 774 to learn more about that.

There are two main kinds of template declarations in C++; class templates and function templates.

Class Templates

Rule: An instantiation of a class template is translated in the same way as the non-template class that is obtained if all formal arguments of the class template declaration are substituted with the actual arguments of the class template instantiation.

This translation rule implies that class template instantiations will become newtypes in SDL. The name of such a newtype will consist of the name of the template class, followed by the names of all actual template arguments of the template instantiation. The name will also be prefixed with a string that by default is "tpl_". The option `-prefix` can be used to configure this string.

Example 131: Translation of class template instantiations ––––––––

C++:

```
template <class T> class C {
public:
        T t;
  T f();
  C(T v);
};
typedef C<int> mytype; // Class template
instantiation
```

SDL:

```
NEWTYPE ptr_tpl_C_int Ref( tpl_C_int);
  OPERATORS
    ptr_tpl_C_int : -> ptr_tpl_C_int;
    ptr_tpl_C_int : ptr_tpl_C_int -> ptr_tpl_C_int;
ENDNEWTYPE ptr_tpl_C_int;EXTERNAL 'C++';
NEWTYPE tpl_C_int /*#REFNAME 'C<int >'*/
  STRUCT
    t int;
  OPERATORS
    tpl_C_int /*#REFNAME 'C'*/ : int -> tpl_C_int;
    f : tpl_C_int -> int;
    tpl_C_int /*#REFNAME 'C'*/ : tpl_C_int ->
tpl_C_int;
ENDNEWTYPE tpl_C_int;EXTERNAL 'C++';
SYNTYPE mytype = tpl_C_int
ENDSYNTYPE mytype;EXTERNAL 'C++';
```

––

Note that a #REFNAME directive passes the C++ name of the class template instantiation to the Code Generator.

Function Templates

Rule: An instantiation of a function template is translated in the same way as the non-template function that is obtained if all formal arguments of the function template declaration are substituted with the actual arguments of the function template instantiation.

This translation rule implies that function template instantiations will become operators in SDL. The name of such an operator will consist of the name of the template function, followed by the names of all actual template arguments of the template instantiation. The name will also be prefixed with a string that by default is "tpl_". The option `-prefix` can be used to configure this string.

Since a function template is instantiated when called, a C++ header file will normally not contain any function template instantiations. Instead an import specification should be used to provide the necessary instantiations (see "Template Instantiations" on page 774). In Example 132 below an import specification is used to instantiate the template function with the type `int*`.

Example 132: Translation of function template instantiations–––––––

C++:

```
template <class T> T func(T t);
```

Import Specification:

```
TRANSLATE {
  func<int*>
}
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    tpl_func_ptr_int /*#REFNAME 'func<int* >'*/ :
ptr_int
-> ptr_int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
NEWTYPE ptr_int Ref( int);
  OPERATORS
    ptr_int : -> ptr_int;
    ptr_int : ptr_int -> ptr_int;
```

```
ENDNEWTYPE ptr_int;EXTERNAL 'C++';
```

As can be seen from the translation of the example above, the #REFNAME directive contains the C++ name of the template instantiation written on the so called explicit form[1]. This makes the Code Generator use this explicit form when the template function is called from SDL.

Note:

Calling a function template from SDL requires that the target C++ compiler can handle calls using the explicit form of the function template instantiation.

Default Template Arguments

Rule: An instantiation of a template declaration with default arguments is translated in the same way as an ordinary template instantiation, where omitted actual arguments in the instantiation are substituted with the specified default types or values.

This translation rule is very similar to the one used for functions with default arguments (see "Default Arguments" on page 788).

Example 133: Translation of templates with default arguments

C++:

```
template <class T, class U = char, int i = 5> class
C {
public:
        T t[i];
  T f();
  C(U p1);
};
C<int> var1; // Using all the default values
C<int, bool> var2; // Using the default value for i
C<int, bool, 5> var3; // Not using any default value
```

SDL:

```
NEWTYPE ptr_tpl_C_int_char_5 Ref( tpl_C_int_char_5);
  OPERATORS
```

1. In the explicit form of a function template instantiation, all actual template arguments are provided explicitly in the instantiation rather than being deduced from the types of the actual arguments in a call to the function template.

```
    ptr_tpl_C_int_char_5 : -> ptr_tpl_C_int_char_5;
    ptr_tpl_C_int_char_5 : ptr_tpl_C_int_char_5 ->
    ptr_tpl_C_int_char_5;
ENDNEWTYPE ptr_tpl_C_int_char_5;EXTERNAL 'C++';
NEWTYPE tpl_C_int_char_5 /*#REFNAME 'C<int, char, 5
>'*/
  STRUCT
    t_arr_5_int;
  OPERATORS
    tpl_C_int_char_5 /*#REFNAME 'C'*/ : char ->
tpl_C_int_char_5;
    f : tpl_C_int_char_5 -> int;
    tpl_C_int_char_5 /*#REFNAME 'C'*/ :
tpl_C_int_char_5
-> tpl_C_int_char_5;
ENDNEWTYPE tpl_C_int_char_5;EXTERNAL 'C++';
DCL var1 tpl_C_int_char_5; EXTERNAL 'C++';
DCL var2 tpl_C_int_bool_5; EXTERNAL 'C++';
NEWTYPE arr_5_int CArray( 5, int);
ENDNEWTYPE arr_5_int;EXTERNAL 'C++';
NEWTYPE ptr_tpl_C_int_bool_5 Ref( tpl_C_int_bool_5);
  OPERATORS
    ptr_tpl_C_int_bool_5 : -> ptr_tpl_C_int_bool_5;
    ptr_tpl_C_int_bool_5 : ptr_tpl_C_int_bool_5 ->
    ptr_tpl_C_int_bool_5;
ENDNEWTYPE ptr_tpl_C_int_bool_5;EXTERNAL 'C++';
NEWTYPE tpl_C_int_bool_5 /*#REFNAME 'C<int, bool, 5
>'*/
  STRUCT
    t_arr_5_int;
  OPERATORS
    tpl_C_int_bool_5 /*#REFNAME 'C'*/ : bool ->
tpl_C_int_bool_5;
    f : tpl_C_int_bool_5 -> int;
-> tpl_C_int_bool_5;
ENDNEWTYPE tpl_C_int_bool_5;EXTERNAL 'C++';
DCL var3 tpl_C_int_bool_5; EXTERNAL 'C++';
```

Note that although the template instantiations of `var2` and `var3` in the example above look different, they evaluate to the same template type both in C++ and in the SDL translation.

Miscellaneous

This section covers some miscellaneous issues that have not been discussed so far. They are divided into constructs that are part of the C or C++ languages, and constructs that are not part of the languages as such, but that nevertheless may be found in an input C/C++ header file.

Language Constructs

Volatile

Rule: A volatile declaration is translated in the same way as an ordinary declaration.

The `volatile` specifier can be looked upon as some kind of compiler directive, and needs therefore not be visible in the SDL translation.

Linkage

Rule: The linkage of a C/C++ identifier is not visible in the SDL translation of the identifier.

There is one important exception to this rule; static linkage of class members affects their translation as described in "Static Members" on page 803.

In general, the linkage of a C/C++ identifier can be specified to be internal or external using the keywords `static` or `extern` (although the former is a deprecated feature in C++ for all declarations but class members). In C++ it is also possible to use the `extern` keyword to specify that a set of declarations have C linkage, i.e. belong to a translation unit that is compiled with a C compiler.

Example 134: Translation of identifiers with different linkage –––––––

C++:

```
extern int a; // Declaration of a
extern int a; // Legal redeclaration of a
int a; // Definition of a
extern "C" {
  struct S {
    int x;
  };
}
```

SDL:

```
DCL a int; EXTERNAL 'C++';
NEWTYPE ptr_S Ref( S);
  OPERATORS
    ptr_S : -> ptr_S;
    ptr_S : ptr_S -> ptr_S;
ENDNEWTYPE ptr_S;EXTERNAL 'C++';
NEWTYPE S
  STRUCT
    x int;
```

```
  OPERATORS
    S : -> S;
    S : S -> S;
ENDNEWTYPE S;EXTERNAL 'C++';
```

Note that the `extern "C"` directive in this example does not affect the mapping of `S` at all. For example, it will be possible to instantiate `S` using the `new` operator.

Non-Language Constructs

Macros

Rule: Macros are not translated to SDL.

The reason for not translating macros is that they are not part of the C or C++ languages. Macros are expanded and removed by the preprocessor before CPP2SDL performs the translation.

Some header files (especially C headers) contain numerous macros that could be useful or even essential to access in SDL. Fortunately most macros can actually be accessed from SDL, although they are not translated by CPP2SDL. Refer to "Constants" on page 794 for more information.

SDL Sorts in C/C++

Rule: A C/C++ type called "SDL_<sort>", where <sort> is a predefined SDL sort, is translated to that SDL sort.

Since this translation rule restricts the way ordinary C/C++ types may be named, it is only respected by CPP2SDL if the `-sdlsorts` option is set.

Example 135: Translation of types referring to SDL sorts

C++:

```
SDL_Real func(SDL_Integer, int);
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    func : Integer, int -> Real;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
```

The feature of referring to SDL sorts from a C/C++ header file may be useful if the header has been designed to be used from SDL exclusively.

Special Translation Rules for C Compilers

CPP2SDL by default assumes that its input is C++ code, and that the target compiler is a C++ compiler. In order to also support C code and C target compilers, the translation rules have to be slightly modified. CPP2SDL does this if the `-c` option is set. The tool then executes in “C mode”.

A general difference is how the external specifier for all SDL declarations will be generated. Normally this specifier is followed by the string ‘C++’ to tell the Code Generator that the declaration was translated from a C++ declaration. In C mode the string will instead be ‘C’.

Example 136: Different external specifier in C mode ––––––––––––––

C++:

```
const int a;
```

SDL:

```
SYNONYM a int = EXTERNAL 'C';
```

––

In C, a struct or a union is not a scope unit, which means that declarations inside a struct or a union should be treated as ordinary declarations. This means that the #REFNAME directive that normally is used for specifying the qualified name of for example a nested struct declaration, will not be printed in C mode.

Instead, a #REFNAME directive will be inserted after the name of newtypes that represent tagged types (i.e. structs, unions and enums). The reason is that C, contrary to C++, does not allow such types to be referenced only with the name of the tag. Another place where the "full" type name is required in C mode is in the C style cast that is generated by means of a #REFNAME directive for `IntToEnum` operators (see “Enumerated Types” on page 783).

Example 137: Differences in translation of structs, unions and enums

C++:

```
struct S {
  int a;
  struct SS {
    int b;
  };
};
```

```
typedef enum E {e1, e2, e3} Etype;
```

SDL:

```
NEWTYPE ptr_SS Ref( SS);
ENDNEWTYPE ptr_SS;EXTERNAL 'C';
NEWTYPE SS /*#REFNAME 'struct SS'*/
  STRUCT
    b int;
ENDNEWTYPE SS;EXTERNAL 'C';
NEWTYPE ptr_S Ref( S);
ENDNEWTYPE ptr_S;EXTERNAL 'C';
NEWTYPE S /*#REFNAME 'struct S'*/
  STRUCT
    a int;
ENDNEWTYPE S;EXTERNAL 'C';
NEWTYPE E /*#REFNAME 'enum E'*/
  LITERALS e1, e2, e3;
  OPERATORS
    IntToEnum /*#REFNAME '(enum E)'*/ : int -> E;
    EnumToInt : E -> int; /*#OP(PY)*/
    ORDERING;
ENDNEWTYPE E;EXTERNAL 'C';
SYNTYPE Etype = E
ENDSYNTYPE Etype;EXTERNAL 'C';
```

Finally, note that memory allocation is done differently in C and C++. This is reflected in SDL by using a different definition of the `Ref` generator when CPP2SDL executes in C mode, where for example the `new`, `delete`, `new_array`, and `delete_array` operators are not present. See "Dynamic Memory Management" on page 825 and "SDL Library for Fundamental C/C++ Types" on page 841 for more information.

SDL Library for Fundamental C/C++ Types

The SDL declarations that are generated by CPP2SDL will normally not be semantically correct on their own. They typically contain several references to SDL sorts that represent fundamental C/C++ types, for example `int`, `char` and `bool`, and type declarators such as pointers (`*`) and arrays (`[]`). The SDL representations of all fundamental C/C++ types and type declarators are defined in a library consisting of a few SDL/PR files. The table below lists these files and their contents.

SDL/PR file	Contents
BasicCTypes.pr *Contains SDL representations of fundamental C types. Also contains representations for an untyped pointer (void*) and the array type declarator ([]).*	SYNTYPE int = Integer ENDSYNTYPE int; SYNTYPE unsigned_int = Integer ENDSYNTYPE unsigned_int; SYNTYPE long_int = Integer ENDSYNTYPE long_int; SYNTYPE unsigned_long_int = Integer ENDSYNTYPE unsigned_long_int; SYNTYPE short_int = Integer ENDSYNTYPE short_int; SYNTYPE unsigned_short_int = Integer ENDSYNTYPE unsigned_short_int; SYNTYPE char = Character ENDSYNTYPE char; SYNTYPE signed_char = Character ENDSYNTYPE signed_char; SYNTYPE unsigned_char = Octet ENDSYNTYPE unsigned_char; SYNTYPE float = Real ENDSYNTYPE float; SYNTYPE double = Real ENDSYNTYPE double; NEWTYPE ptr_void LITERALS Null; DEFAULT Null; ENDNEWTYPE ptr_void; GENERATOR CArray (CONSTANT Length, TYPE Itemsort) OPERATORS modify!: CArray, Integer, Itemsort -> CArray; extract!: CArray, Integer -> Itemsort; ENDGENERATOR CArray;

SDL/PR file	Contents
BasicC++Types.pr *Contains SDL representations of fundamental C++ types. Also contains operators representing implicit constructors for fundamental types.* *Note that this file includes the files BasicCtypes.pr and ExtraCTypes.pr.*	/*#INCLUDE 'BasicCTypes.pr'*/ /*#INCLUDE 'ExtraCTypes.pr'*/ SYNTYPE bool = Boolean ENDSYNTYPE bool; NEWTYPE wchar_t ENDNEWTYPE wchar_t; NEWTYPE __ConstructorOperators /*#NOTYPE*/ OPERATORS int: -> int; int: int -> int; unsigned_int: -> unsigned_int; unsigned_int: unsigned_int -> unsigned_int; long_int: -> long_int; long_int: long_int -> long_int; unsigned_long_int: -> unsigned_long_int; unsigned_long_int: unsigned_long_int -> unsigned_long_int; short_int: -> short_int; short_int: short_int -> short_int; unsigned_short_int: -> unsigned_short_int; unsigned_short_int: unsigned_short_int -> unsigned_short_int; char: -> char; char: char -> char; signed_char: -> signed_char; signed_char: signed_char -> signed_char; unsigned_char: -> unsigned_char; unsigned_char: unsigned_char -> unsigned_char; float: -> float; float: float -> float; double: -> double; double: double -> double; ptr_void: -> ptr_void; ptr_void: ptr_void -> ptr_void; bool: -> bool; bool: bool -> bool; wchar_t: -> wchar_t; wchar_t: wchar_t -> wchar_t; ENDNEWTYPE __ConstructorOperators;EXTERNAL 'C++';

SDL/PR file	Contents
ExtraCTypes.pr *Contains SDL representations of additional fundamental C types.*	SYNTYPE long_long_int = Integer ENDSYNTYPE long_long_int; SYNTYPE unsigned_long_long_int = Integer ENDSYNTYPE unsigned_long_long_int;
ExtraC++Types.pr *Contains SDL representations of additional fundamental C++ types.*	NEWTYPE__ExtraConstructorOperators long_long_int : -> long_long_int; long_long_int : long_long_int -> long_long_int; unsigned_long_long_int : -> unsigned_long_long_int; unsigned_long_long_int : unsigned_long_long_int - > unsigned_long_long_int ENDNEWTYPE__ExtraConstructorOperators
CPointer.pr *Contains SDL representation of the C pointer type declarator (*).*	GENERATOR Ref (TYPE Itemsort) LITERALS Null, Alloc; OPERATORS modify! : Ref, Integer, Itemsort -> Ref; extract! : Ref, Integer -> Itemsort; "*>" : Ref, Itemsort -> Ref; "*>" : Ref -> Itemsort; "&" : Itemsort -> Ref; make! : Itemsort -> Ref; free : in/out Ref; "+" : Ref, Integer -> Ref; "-" : Ref, Integer -> Ref; cast : Ref -> ptr_void; cast : ptr_void -> Ref; DEFAULT Null; ENDGENERATOR Ref;

SDL/PR file	Contents
C++Pointer.pr *Contains SDL representation of the C++ pointer type declarator (*).*	GENERATOR Ref (TYPE Itemsort) LITERALS Null; OPERATORS modify! : Ref, Integer, Itemsort -> Ref; extract! : Ref, Integer -> Itemsort; "*>" : Ref, Itemsort -> Ref; "*>" : Ref -> Itemsort; "&" : Itemsort -> Ref; new : Itemsort -> Ref; delete : Ref; new_array : Itemsort, Integer -> Ref; delete_array : Ref; "+" : Ref, Integer -> Ref; "-" : Ref, Integer -> Ref; cast : Ref -> ptr_void; cast : ptr_void -> Ref; DEFAULT Null; ENDGENERATOR Ref;

If the option `-generatecpptypes` is set, CPP2SDL will include some of the files from the table above in the SDL translation. Which files that are included depends on if CPP2SDL executes in C or C++ mode (controlled by the `-c` option).

The following files will be included in C mode:

- `BasicCTypes.pr`
- `CPointer.pr`

The following files will be included in C++ mode:

- `BasicC++Types.pr`
- `C++Pointer`

The reason for breaking out the types `long long int` and `unsigned long long int` into separate files, is that not all compilers support these types. These files must be manually included if these types are present in the input headers.

Hint:

The syntype definitions of the SDL sorts that represent fundamental C/C++ types can easily be changed. For example, the definitions of the SDL sorts ‘char’ and ‘unsigned char’ could be swapped if the target platform specifies that a simple ‘char’ is unsigned rather than signed.

Example usage of some C/C++ functionality

Overloaded Operators

This example illustrates how to call an operator which has been made accessible by CPP2SDL. There are two operators defined and used, the first being a member operator, and the second a non-member operator. Also see “Overloaded Operators” on page 846.

C++:

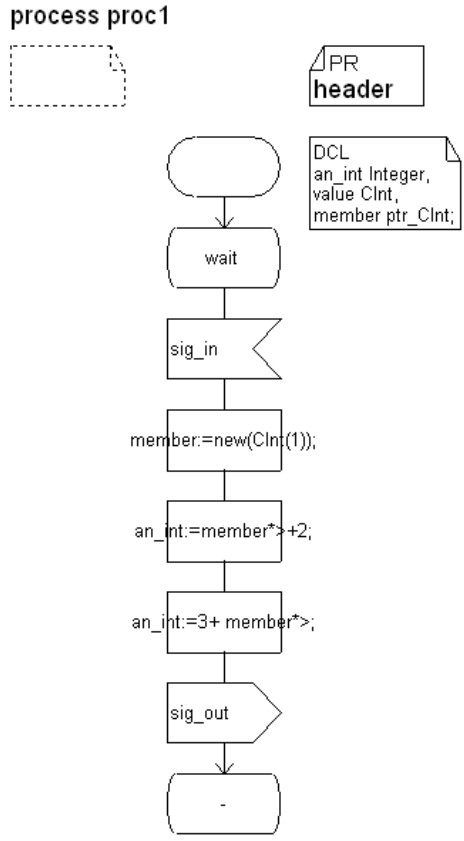
```
class CInt {
  int val;
        public:
  CInt() : val(0) {};
  CInt(int i) : val(i) {};
  int value() const { return val; };

  int operator+(const int& i){val+= i; return val;};
};

int operator+(const int& left, const CInt& right)
{return right.value()+left;};
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    "+" /*#REFNAME 'operator+'*/ : int, CInt -> int;
ENDNEWTYPE global_namespace;EXTERNAL 'C++';
/*#SDTREF(TEXT,header_CPP2SDL.i,16,7)*/
NEWTYPE ptr_CInt Ref( CInt);
  OPERATORS
    ptr_CInt : -> ptr_CInt;/* implicit parameter-
less constructor */
    ptr_CInt : ptr_CInt -> ptr_CInt;/* implicit copy
constructor */
ENDNEWTYPE ptr_CInt;EXTERNAL 'C++';
/*#SDTREF(TEXT,header_CPP2SDL.i,16,7)*/
NEWTYPE CInt
```

```
  OPERATORS
    "+" /*#REFNAME 'operator+'*/ : CInt, int -> int;
    CInt : -> CInt;
    CInt : int -> CInt;
    value : CInt -> int; /*#CONSTANT*/
    CInt : CInt -> CInt;/* implicit copy constructor
*/
ENDNEWTYPE CInt;EXTERNAL 'C++';
```

Use in SDL:



String handling

It is possible to do C-style string handling in SDL, by using the standard C header **`string.h`**. By including '`string.h`' and '`stdio.h`' we are given access to the functions defined within them in SDL. You may notice that strcpy is defined in the hand written header as well as in '`string.h`'. The former definition allows us to assign the string "goodbye" to empty, without using the return value of strcpy, and importing it to SDL. Also needed is an allocate and deallocate function. An example allocate function has been defined in the header, a deallocate function should also be done in the practice to avoid memory leaks.

C:

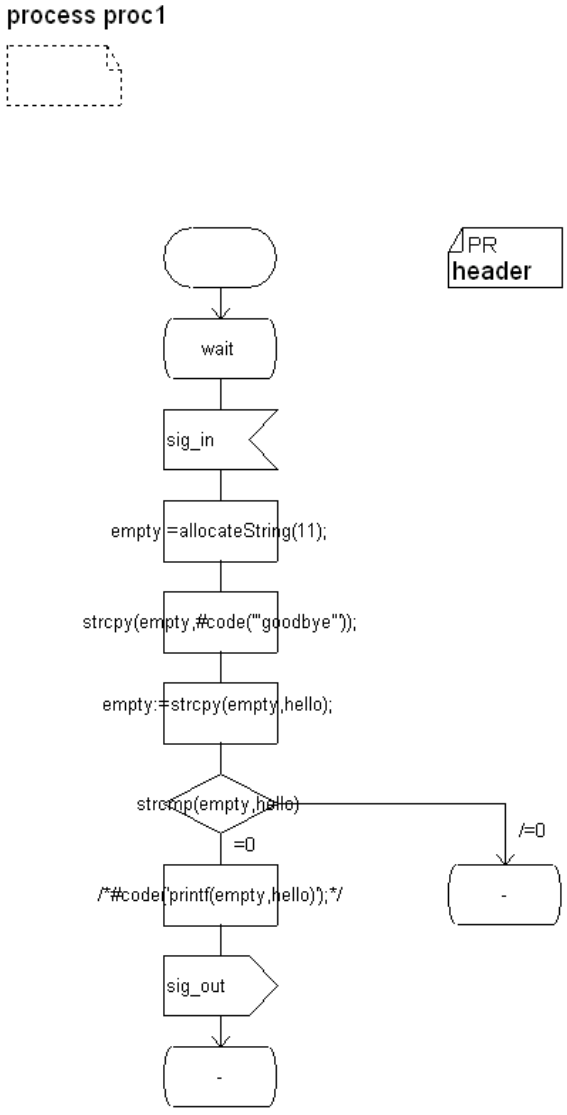
```
#include<string.h>
#include<stdio.h>

#ifdef __CPP2SDL__
void strcpy(char*,char*);
#endif

typedef char* string;
char ara[10];
string hello= "hello";
string empty;

char* allocateString(int length) {
  return (char*) calloc(length,sizeof(char));
}
```

SDL:

```
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    memcpy : ptr_void, ptr_void, size_t -> ptr_void;
    memcmp : ptr_void, ptr_void, size_t -> int;
    memset : ptr_void, int, size_t -> ptr_void;
    _strset : ptr_char, int -> ptr_char;
    strcpy : ptr_char, ptr_char -> ptr_char;
    strcat : ptr_char, ptr_char -> ptr_char;
    strcmp : ptr_char, ptr_char -> int;
    strlen : ptr_char -> size_t;
    ....
    unlink : ptr_char -> int;
    strcpy : ptr_char, ptr_char;
    allocateString : int -> ptr_char;
ENDNEWTYPE global_namespace;EXTERNAL 'C';
...
SYNTYPE string = ptr_char
ENDSYNTYPE string;EXTERNAL 'C';
/*#SDTREF(TEXT,header_CPP2SDL.i,728,6)*/
NEWTYPE arr_10_char CArray( 10, char);
```

```
ENDNEWTYPE arr_10_char;EXTERNAL 'C';
/*#SDTREF(TEXT,header_CPP2SDL.i,728,6)*/
DCL ara arr_10_char; EXTERNAL 'C';
/*#SDTREF(TEXT,header_CPP2SDL.i,729,8)*/
DCL hello string; EXTERNAL 'C';
/*#SDTREF(TEXT,header_CPP2SDL.i,730,8)*/
DCL empty string; EXTERNAL 'C';
```

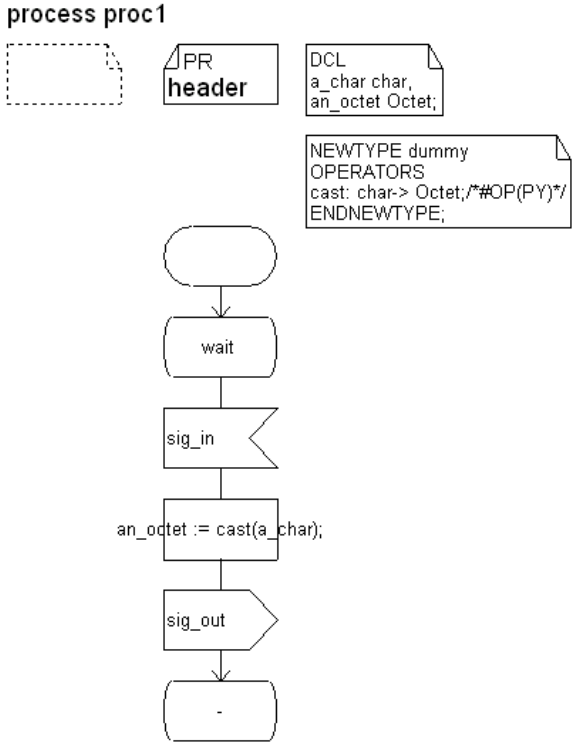
Use in SDL:


process proc1
PR
header
wait
sig_in
empty :=allocateString(11);
strcpy(empty,#code("goodbye"));
empty:=strcpy(empty,hello);
strcmp(empty,hello)
=0
/=0
/*#code(printf(empty,hello));*/
-
sig_out
-

Type conversion

Some type conversions are easier in C/C++ than in SDL, in particular those that are implicit. An implicit type conversion must often be explicit in SDL, by introducing a simple cast operator that performs the conversion. For example, by using the mapping of SDL sorts between unsigned_char and char we can create a cast operator in SDL that converts a char to an Octet. (See "Fundamental Types" on page 779 for more information). This corresponds to the implicit conversion in C/C++ between char and unsigned char.

Use in SDL:



Error Handling

The input headers to CPP2SDL have in many cases been compiled with a C/C++ compiler previously, and it should then be relatively uncommon that CPP2SDL will have to issue any error messages. However, differences in language support, and inappropriate preprocessor settings, are common sources for error reports also from input files that otherwise are perfectly correct.

If CPP2SDL finds an error in the input, a message will be printed that briefly describes the reason for the error.

> **Note:**
>
> The error messages produced by CPP2SDL are often less descriptive than the corresponding error messages from a C/C++ compiler. If CPP2SDL reports errors in a header file, it is therefore a good idea to run a C/C++ compiler on the same header file to get more information about the reason for the error.

The format of printed error messages are described in "Source and Error References" on page 776.

CPP2SDL performs a complete syntactic analysis of the input C/C++ code, and syntax errors are reported as shown in Example 138 below.

Example 138: Syntax errors ––––––––––––––––––––––––––––––––

File syn.h:

```
int f(};
conts int i = 7;
```

Command Prompt:

```
% cpp2sdl syn.h
Parsing C/C++ input...
Syntax errors found. Cannot perform SDL translation.
#SDTREF(TEXT,syn.h,1,7)
ERROR 3200 Syntax error.
#SDTREF(TEXT,syn.h,2,7)
ERROR 3200 Syntax error.
2 errors and 0 warnings.
```

––

CPP2SDL will proceed with semantic analysis only if no errors were found during the syntactic analysis. The semantic analysis that is done

by CPP2SDL is not complete according to the C/C++ standards, but only a limited number of semantic tests are performed:

- The type of variables, constants, typedefs, functions, function arguments, actual template arguments, and base classes are checked. If a type is undeclared, or if it depends on an undeclared type[1], an error message will be issued.
- The actual arguments of a template instantiation are checked against the formal arguments of the template definition. If there are too few or too many actual arguments, or if there are type mismatches between actual and formal arguments, an error message will be issued.

Example 139: Semantic errors ––––––––––––––––––––––––––––––

File sem.h:

```
template <class U, int d> class S {
public:
  U arr[d];
};
typedef unknown T; // T depends on undeclared type
const unknown a = 3;
T f(int, char);
S<T, 3> var;
typedef S<> t1; // Too few actual arguments
typedef S<char, 5, 5> t2; // Too many actual
arguments
typedef S<int, int> t3; // Argument type mismatch
```

Command Prompt:

```
% cpp2sdl -errorlimit 10 sem.h
Parsing C/C++ input...
Translating C/C++ to SDL...
Generating SDL...
#SDTREF(TEXT,sem.h,11,9)
ERROR 3263 Illegal instantiation of template 'S'.
#SDTREF(TEXT,sem.h,10,9)
ERROR 3263 Illegal instantiation of template 'S'.
#SDTREF(TEXT,sem.h,9,9)
ERROR 3263 Illegal instantiation of template 'S'.
#SDTREF(TEXT,sem.h,8,3)
ERROR 3261 The type 'T' is undeclared, or is
depending on an undeclared type.
#SDTREF(TEXT,sem.h,7,3)
ERROR 3261 The type 'T' is undeclared, or is
```

1. One example of such a type dependency is when the source type of a typedef type is undeclared. Usages of the typedef type will then be considered to be undeclared.

```
depending on an undeclared type.
#SDTREF(TEXT,sem.h,6,15)
ERROR 3261 The type 'unknown' is undeclared, or is
depending on an undeclared type.
#SDTREF(TEXT,sem.h,5,17)
ERROR 3261 The type 'unknown' is undeclared, or is
depending on an undeclared type.
#SDTREF(TEXT,sem.h,1,33)
WARNING 3211 Cannot translate template declaration.
The declaration will be ignored.
7 errors and 1 warnings.
```

Note that the command-line option `-errorlimit` can be used to set a limit for the number of errors to report before terminating a translation.

CPP2SDL Messages

CPP2SDL may produce three kinds of messages during the translation of a set of header files.

- Error messages are printed if CPP2SDL finds any syntactic or semantic errors in the input header files. See "Example usage of some C/C++ functionality" on page 846 for more information about how CPP2SDL handles errors in the input.
- Warnings are given if CPP2SDL finds language constructs that for some reason cannot be fully translated. The tool also prints warnings if it has to make assumptions about a construct that not necessarily are valid.
- Information messages are all other messages that are printed.

The rest of this section lists and explains all errors and warnings that may be issued by CPP2SDL.

Errors

`ERROR 3200 Syntax error.`

An error was found during the syntactic analysis of the input. CPP2SDL will not continue with semantic analysis and translation to SDL, since the program is not correct.

> **Note:**
> If this error message is printed for a program that is accepted by a C/C++ compiler, make sure that the correct language dialect has been set to CPP2SDL by means of the `-dialect` option.

`ERROR 3260 The identifier <identifier name> is undeclared.`

An identifier is undeclared, i.e. the program is not semantically correct and will therefore not be translated to SDL.

`ERROR 3261 The type <type name> is undeclared, or is depending on an undeclared type.`

A type is undeclared, or depends on a type that is undeclared. A type defined by a typedef of an undeclared type is an example of a type that depends on a undeclared type. Since the program is not semantically correct, it will not be translated to SDL.

```
ERROR 3262 The base <base class name> is undeclared.
```

A class inherits from a base class that is undeclared. This is a semantic error, and the program will thus not be translated to SDL.

```
ERROR 3263 Illegal instantiation of template <template
name>.
```

A template instantiation is semantically incorrect. Make sure that the number of actual arguments in the template instantiation matches the number of formal arguments in the template declaration, and that the kinds of the arguments are correct. Since the program is not semantically correct, it will not be translated to SDL.

Warnings

```
WARNING 3201 Static member variable <variable name> will
not be globally accessible since no SDL variables are al-
lowed.
```

A static member variable cannot be fully translated, since no external variables are allowed in the context where the generated SDL declarations are to be injected. The static variable will still be accessible as an ordinary member variable, but not as a globally accessible variable. Unset the option `-novariables` to allow generation of external SDL variables.

```
WARNING 3202 Static overloaded operator <operator name>
will not be globally accessible.
```

A static overloaded operator cannot be fully translated, since it is not possible to qualify the name of an overloaded operator in SDL which otherwise would be required. The static operator will still be accessible as an ordinary member overloaded operator, but not as a globally accessible overloaded operator. Refer to "Overloaded Operators" on page 828.

```
WARNING 3203 Cannot translate incomplete type declara-
tion without declared objects. The declaration will be
ignored.
```

An incomplete type declaration that is not used as the type of at least one object (e.g. variable, constant, or type) is a useless declaration that will not be translated to SDL. See "Incomplete Types" on page 823 for more about useless incomplete type declarations.

```
WARNING 3204 Cannot translate overloaded operator, since
it is declared in a namespace.
```

An overloaded operator declared in a namespace cannot be translated to SDL, since it is not possible to qualify the name of an overloaded operator in SDL which otherwise would be required. Refer to "Overloaded Operators" on page 828.

```
WARNING 3205 Cannot translate overloaded shift operator,
since the '<' or '>' operator also is overloaded in this
scope.
```

The translation rule for overloaded operators only supports translation of either the `<` and `>` operators or the `<<` and `>>` operators. This warning is given if an operator from both these operator pairs are overloaded in a certain scope. See "Overloaded Operators" on page 828 for more information.

```
WARNING 3206 Cannot translate overloaded operator, since
no corresponding SDL operator exists.
```

An overloaded operator cannot be translated to SDL, since no appropriate SDL operator exists that could represent it. The table in "Overloaded Operators" on page 828 shows what overloaded C++ operators that may be represented in SDL.

```
WARNING 3207 Unable to evaluate sizeof expression prop-
erly.
```

A constant expression contains a usage of the `sizeof()` operator, and could therefore not be safely evaluated by CPP2SDL. The translation of the constant expression may thus be incorrect, and should be manually reviewed. See "Constant Expressions" on page 795 for more information about constant expressions.

```
WARNING 3208 The member <member name> of <class name> in-
herited via <base class names> is inaccessible and will
not be translated.
```

An inherited class member cannot be accessed in C++ due to a combination of multiple inheritance and base classes with members having the same name. The member will thus not be translated to SDL.

```
WARNING 3209 Cannot translate function pointer type. It
will be represented by ptr_void.
```

This warning is given when a function pointer type is encountered in the input. The support for function pointers is limited (see "Function Pointers" on page 789), and they will be represented as untyped pointers in SDL (i.e. `ptr_void`).

```
WARNING 3210 Cannot translate typedef of function type.
The declaration will be ignored.
```

A typedef declaration where the source type is a function type cannot be translated to SDL, since there is no translation rule for function types.

```
WARNING 3211 Cannot translate template declaration. The
declaration will be ignored.
```

A template declaration cannot be translated to SDL. Only instantiations of a template declaration can be translated. Note that this warning is given also when a template instantiation has been specified in an import specification (see "Template Instantiations" on page 774). In that case the warning could be ignored.

```
WARNING 3212 Cannot fully translate ellipsis function.
Unspecified function arguments will be ignored.
```

A function with unspecified arguments (a.k.a. an ellipsis function) cannot be fully translated to SDL, since no information has been provided about the unspecified arguments. The function will be translated, but without taking the unspecified arguments into consideration. See "Prototypes for Ellipsis Functions" on page 774 to learn how to use an import specification to provide CPP2SDL with actual arguments for unspecified formal arguments of ellipsis functions.

```
WARNING 3213 The typedef name <name> conflicts with the
name of another non-compatible type. The declaration
will be ignored.
```

A typedef declaration cannot be translated to SDL, since the typedef name is the same as another type that is not type compatible with the type defined by the typedef itself. This warning may be given for typedefs of pointers or arrays of tagged types. For example:

```
typedef struct T {
  int i;
} *T;
```

This declaration, which is illegal in C++ but legal in C, contains two types called `T` that are type incompatible. CPP2SDL will make the type `struct T` available in SDL (called `T` there), while the type `T` will not be translated.

It is recommended to change the name of either the typedef name or the type tag to enable CPP2SDL to translate both types, and thus avoid getting this warning.

```
WARNING 3290 The identifier <identifier name> does not
refer to a declared object. It will be ignored.
```

This warning is given if CPP2SDL finds an identifier in an import specification that does not exist in the input program. The identifier will be ignored.

```
WARNING 3291 Cannot translate the identifier <identifier
name>, since it is a class member.
```

This warning is given if CPP2SDL finds an identifier in an import specification that refers to a class member in the input program. The identifier will not be translated, since only declarations in namespaces may be translated (see "Import Specifications" on page 771).

Chapter 15 The CPP2SDL Tool